Programming Language Semantics

A Rewriting Approach

Grigore Roșu

University of Illinois at Urbana-Champaign
Contents

1 Introduction 7

2 Background and Preliminaries 15
   2.1 Mathematical Background ............................................. 16
       2.1.1 Functions .......................................................... 16
       2.1.2 Partial Functions ................................................ 17
       2.1.3 Context-Free Grammars and the Mixfix Algebraic Notation .... 18
       2.1.4 Basic Term Rewriting .............................................. 19
       2.1.5 On Logics and Logic Deduction ................................... 20
   2.2 Basic Computability Elements .......................................... 22
       2.2.1 Turing Machines ................................................... 22
       2.2.2 Universal Machines, the Halting Problem, and Decision Problems 24
       2.2.3 The Arithmetic Hierarchy ........................................ 25
       2.2.4 Notes ................................................................. 26
       2.2.5 Exercises ........................................................... 26
   2.3 Propositional Logic ....................................................... 27
       2.3.1 Syntax and Semantics .............................................. 27
       2.3.2 Proof System and Soundness ..................................... 28
       2.3.3 Consistency and Model Existence ................................. 29
       2.3.4 Completeness and Compactness ................................ 30
       2.3.5 Notes ................................................................. 30
       2.3.6 Exercises ........................................................... 31
   2.4 Equational Logic .......................................................... 32
       2.4.1 Signatures, Algebras, Equations, Satisfaction .................. 32
       2.4.2 Proof System and Soundness ..................................... 32
       2.4.3 Computation as Equational Deduction ............................ 33
       2.4.4 Initial Algebra ..................................................... 36
       2.4.5 Completeness ........................................................ 37
       2.4.6 Defining Data-Types Equationally ............................... 42
       2.4.7 Notes ................................................................. 49
       2.4.8 Exercises ........................................................... 49
   2.5 Rewrite logic .............................................................. 50
       2.5.1 Equations versus Rewrite Rules .................................. 50
       2.5.2 Proof System and Soundness ..................................... 55
       2.5.3 Computation as Rewrite Logic Deduction ........................ 58
### 2.5.4 Completeness

Rewrite Logic as a Universal Logical Formalism

Maude: A High Performance Rewrite Logic System

Exercises

Notes

### 2.6 Complete Partial Orders and the Fixed-Point Theorem

Posets, Upper Bounds and Least Upper Bounds

Complete Partial Orders

Monotone and Continuous Functions

The Fixed-Point Theorem

Equational Framework for Fixed-Points

Exercises

Notes

### 2.7 Category Theory

3 Executable Semantics Approaches

3.1 IMP: A Simple Imperative Language

3.1.1 IMP Syntax

3.1.2 IMP State

3.1.3 Notes

3.2 Big-Step Structural Operational Semantics (Big-Step SOS)

3.2.1 IMP Configurations for Big-Step SOS

3.2.2 The Big-Step SOS Rules of IMP

3.2.3 Big-Step SOS in Rewrite Logic

3.2.4 Defining a Type System for IMP Using Big-Step SOS

3.2.5 Notes

3.2.6 Exercises

3.3 Small-Step Structural Operational Semantics (Small-Step SOS)

3.3.1 IMP Configurations for Small-Step SOS

3.3.2 The Small-Step SOS Rules of IMP

3.3.3 Small-Step SOS in Rewrite Logic

3.3.4 Notes

3.3.5 Exercises

3.4 Denotational Semantics

3.4.1 The Denotational Semantics of IMP

3.4.2 Denotational Semantics in Equational/Rewrite Logic

3.4.3 Notes

3.4.4 Exercises

3.5 IMP++: IMP Extended with Several Features

3.5.1 Adding Variable Increment

3.5.2 Adding Input/Output

3.5.3 Adding Abrupt Termination

3.5.4 Adding Dynamic Threads

3.5.5 Adding Local Variables

3.5.6 Putting Them All Together: First Attempt

3.5.7 Putting Them All Together: Second Attempt

4
3.5.8 Notes ................................................................. 204
3.5.9 Exercises ......................................................... 205

3.6 Modular Structural Operational Semantics (MSOS) .............................. 213
  3.6.1 The MSOS of IMP ............................................... 218
  3.6.2 The MSOS of IMP++ ............................................ 218
  3.6.3 MSOS in Rewrite Logic ....................................... 224
  3.6.4 Notes .............................................................. 240
  3.6.5 Exercises ......................................................... 230

3.7 Reduction Semantics with Evaluation Contexts ................................. 233
  3.7.1 The Reduction Semantics with Evaluation Contexts of IMP ............. 236
  3.7.2 The Reduction Semantics with Evaluation Contexts of IMP++ ........ 239
  3.7.3 Reduction Semantics with Evaluation Contexts in Rewrite Logic ...... 242
  3.7.4 Notes .............................................................. 263
  3.7.5 Exercises ......................................................... 264

3.8 The Chemical Abstract Machine (CHAM) ......................................... 269
  3.8.1 The CHAM of IMP ............................................... 272
  3.8.2 The CHAM of IMP++ ............................................ 278
  3.8.3 CHAM in Rewrite Logic ....................................... 281
  3.8.4 Notes .............................................................. 291
  3.8.5 Exercises ......................................................... 293

3.9 Advanced Denotational Semantics: Continuations, Monads and Powerdomains 295

3.10 Discussion: Advantages, Disadvantages and Comparisons .................. 296

3.11 Existing Approaches and Their Limitations for Programming Language Semantics 296
  3.11.1 Structural Operational Semantics (SOS) Frameworks .................. 296
  3.11.2 Rewriting Frameworks ......................................... 297
  3.11.3 old limitations stuff about big-step and small-step SOS ............ 299
  3.11.4 old stuff about MSOS ......................................... 303
  3.11.5 old stuff about context reduction ................................ 303
  3.11.6 other old stuff about context reduction .......................... 305
  3.11.7 older stuff about cham ....................................... 308

3.12 K ................................................................. 309

3.13 Notes .............................................................. 310

4 Idealized Programming Languages, Features, and Models of Computation 311
  4.1 Combinatory Logic .................................................. 312
  4.2 Turing Machines ................................................... 317
    4.2.1 Notes .......................................................... 317
  4.3 Post Correspondence ................................................ 318
  4.4 Counter Machines .................................................. 319
  4.5 Lambda Calculus .................................................... 320
    4.5.1 Syntax .......................................................... 320
    4.5.2 Free and Bound Variables .................................... 321
    4.5.3 Substitution .................................................... 321
    4.5.4 Alpha Conversion .............................................. 322
    4.5.5 Beta Equivalence and Beta Reduction .......................... 322
    4.5.6 Lambda Calculus as a Programming Language .................... 324
take into account Peter’s comments on both my lecture notes and the
previous draft on K
use either rewrite logic or rewriting logic everywhere
Chapter 1

Introduction

This paper is an informal introduction to K, a rewriting-based language definitional framework. By saying that K is a “rewriting-based” framework, in contrast to a “reduction-based” one, we mean that in K rules can be applied concurrently and unrestricted by context, following the basic intuitions and operational/semantical strengths of Meseguer’s rewrite logic [42]. K was first introduced by the author in an algebraic form in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign in Fall 2003 [64], as a means to define executable concurrent languages in rewrite logic using the Maude executable specification system [17]. A more formal, detailed algebraic description of K can be found in [66].

The major purpose of this report is to explain K using a non-algebraic, intuitive and conventional notation based on context-free grammars enriched with structural equivalences and rewrite rules, as opposed to algebraic signatures, algebraic specifications and rewrite logic theories. The reason for doing so is that we strongly believe that K can also be useful to the working programming language or calculi designer who may not like, may not be familiar with, or simply may not want to use the algebraic notation. An additional purpose for stripping K of its algebraic armor is to reflect the important fact that K is more than a technique to implement interpreters in Maude; it is a language definitional style, like structural operational semantics (SOS) [60, 61], modular SOS (MSOS) [53], reduction semantics with evaluation contexts (context reduction) [25], or the chemical abstract machine (Cham) [9].

Figure 1 shows definitions of a simple imperative language and of some \( \lambda \)-calculi using both context reduction and K. For a fair comparison on the imperative language, we assumed that both definitions initiate the evaluation of a program \( p \) by wrapping it into double square brackets, \( \llbracket p \rrbracket \); then each definition takes that and initiates the configuration in its own way. The syntax of the simple imperative language is:

\[
\begin{align*}
Var & ::= \text{standard identifiers} \\
Val & ::= \text{Int} \\
AExp & ::= Var | Val | AExp + AExp \\
BExp & ::= \text{true} | \text{false} | AExp \leq AExp \land BExp \land \not BExp \\
Stmt & ::= Var ::= AExp | Stmt; Stmt | \text{if} BExp \text{ then Stmt else Stmt} | \\
& \quad \text{while} BExp \text{ do Stmt} | \text{halt} AExp \\
Pgm & ::= AExp | Stmt; AExp
\end{align*}
\]

To make it more interesting, we assumed that \( + \) is strict but evaluates its arguments non-deterministically (i.e., it may interleave execution steps in its argument expressions), that \( \leq \) is strict from left to right (it first evaluates its first argument completely then its second), and that \( \text{and} \) shortcuts (it evaluates its first argument and it continues to evaluate the second only if the first is true). Only arithmetic expressions can be
A simple imperative language

<table>
<thead>
<tr>
<th>Context reduction</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration and initialization</td>
<td>KResult := Val</td>
</tr>
<tr>
<td></td>
<td>K := KResult</td>
</tr>
<tr>
<td></td>
<td>Config := Int</td>
</tr>
<tr>
<td></td>
<td>[ p ] \rightarrow [p, \theta]</td>
</tr>
<tr>
<td>Cxt := \square</td>
<td>Cxt, State</td>
</tr>
<tr>
<td>Variable lookup</td>
<td>\langle x \rightarrow_k (\sigma)_{\text{state}}</td>
</tr>
<tr>
<td>[ c, \sigma ][x] \rightarrow [ c, \sigma ][\sigma[x]]</td>
<td></td>
</tr>
<tr>
<td>Operators</td>
<td>not BExp</td>
</tr>
<tr>
<td></td>
<td>not Cxt</td>
</tr>
<tr>
<td></td>
<td>Cxt := \ldots</td>
</tr>
<tr>
<td></td>
<td>true and b \rightarrow b</td>
</tr>
<tr>
<td></td>
<td>false and b \rightarrow false</td>
</tr>
<tr>
<td>Statements</td>
<td>BExp and BExp</td>
</tr>
<tr>
<td>Stmt ::= \ldots</td>
<td>skip</td>
</tr>
<tr>
<td>Cxt ::= \ldots</td>
<td>Var := AExp</td>
</tr>
<tr>
<td></td>
<td>[ c, \sigma ][x := v] \rightarrow [ c, \sigma[v/x]][\text{skip}]</td>
</tr>
<tr>
<td></td>
<td>\langle x := v \rightarrow_k (\sigma)_{\text{state}}</td>
</tr>
<tr>
<td></td>
<td>\sigma_{\text{state}}[v/x]</td>
</tr>
<tr>
<td>Cxt ::= \ldots</td>
<td>Cxt; Stmt</td>
</tr>
<tr>
<td>skip; s \rightarrow s</td>
<td></td>
</tr>
<tr>
<td>while b do s \rightarrow</td>
<td></td>
</tr>
<tr>
<td>c[\text{halt} i] \rightarrow i</td>
<td></td>
</tr>
<tr>
<td>Programs</td>
<td>halt AExp</td>
</tr>
<tr>
<td>Cxt ::= \ldots</td>
<td>Cxt; AExp</td>
</tr>
<tr>
<td>skip; a \rightarrow a</td>
<td></td>
</tr>
</tbody>
</table>

\(\lambda\)-calculus

| Cxt ::= \lambda Var.Cxt | Cxt Exp | Exp Cxt |
| (\lambda x.e) e' \rightarrow e[e'/x] |
| (\lambda x.e) e' \rightarrow e[e'/x] |

Call-by-value \(\lambda\)-calculus

| Val ::= \lambda Var.Exp |
| Cxt ::= Cxt Exp | Exp Cxt |
| (\lambda x.e) v \rightarrow e[v/x] |
| \langle(\lambda x.e) v \rightarrow_k (\sigma)_{\text{state}} |
| \langle e[v/x] \rightarrow_k |

Call-by-name \(\lambda\)-calculus

| Cxt ::= Cxt Exp |
| \langle(\lambda x.e) e' \rightarrow_k (\sigma)_{\text{state}} |
| \langle e[e'/x] \rightarrow_k |

Figure 1.1: A simple imperative language and \(\lambda\)-calculi defined using evaluation contexts and K (\(p \in \text{Pgm}, v \in \text{Val}, c \in \text{Cxt}, x \in \text{Var}, \sigma \in \text{State}, i, i_1, i_2 \in \text{Int}, b \in \text{Bool}, s, s_1, s_2 \in \text{Stmt}, a \in \text{AExp}; e, e' \in \text{Exp}\)
assigned, so in particular variables can take only integer values. Programs consist of a statement followed
by an arithmetic expression, so they can only evaluate to an integer value. Halt also halts programs only
with integer values. Booleans can be introduced in programs only as results of comparisons and can be
used in conditionals and loops. This language is admittedly trivial and limited, but it serves our purpose
of introducing K by comparison with context reduction. This language will be non-trivially modified and
extended in Section ??.

Note that even though we had to rewrite the syntactic constructs in the K definition in Figure 1 in order to augment them with strictness attributes\(^1\) the K definition was still more compact than the context reduction one. For the sake of completeness, in both definitions we also included declarations of
default items that appear in almost any language definition and so deserve to be implicit to the framework,
such as the \(\square\) in context reduction, and the configuration constructs \((K)_k\) and \((\text{Set}(\text{Config}))_\top\), respectively,
in K. In general, K definitions tend to be more compact than their equivalent context reduction definitions;
that holds true especially for more complex definitions of languages. As discussed in Section ??, context
reduction definitions can be automatically converted into equivalent K definitions. However, K has several
other advantages over context reduction that may make one consider using K directly at its full strength,
instead of adopting a methodological fragment of it.

Since K makes use only of standard, context-insensitive term rewriting modulo equations, it can be
executed on rewrite engines like Maude almost as is. Nevertheless, the relationship between K and Maude is
like that between SOS, or any of the above-mentioned styles, and Maude (or ML, or Scheme, or Haskell, etc.):
Maude can be used as a means to execute and analyze K language definitions. The interested reader is referred
to [74] for a detailed discussion on how the various language definitional styles can be faithfully embedded
in rewrite logic and then executed using systems like Maude, or even more conventional programming
languages. The basic idea of these faithful embeddings is that a language definition in any of these styles, say
\(L\), can be regarded as a rewrite logic theory, say \(R_L\), in such a way that there is a one-to-one computational
equivalence between reductions using \(L\) and rewrites using \(R_L\). Note that this is significantly stronger than
encoding, or implementing, a framework into another framework: \(R_L\) is \(L\) not an encoding of it, the only
difference between the two being insignificant notational/syntactic conventions. This is totally different from
encoding \(L\) on a Turing machine or in a \(\lambda\)-calculus, for example, because such encodings would not preserve
the intended computational granularity of \(L\)’s reductions (if correct, they would only preserve the relational
behavior of \(L\): whatever is reduced with \(L\) can also be reduced, in any number of steps, with its encodings).

One can naturally ask the following question then:

\[\text{What is the need for yet another language definitional framework that can be embedded in \(K\) in this case, if rewriting logic is already so powerful?}\]

Unfortunately, in spite of its definitional strength as a computational logic framework, rewrite logic does not
give, and does not intend to give, the programming language designer any recipe on how to define a language.
It essentially only suggests the following: however one wants to formally define a programming language
or calculus, one can probably also do it in rewrite logic following the same intuitions and style. Therefore,
rewrite logic can be regarded as a meta-framework that supports definitions of programming languages and
calculi among many other things, providing the language designer with a means to execute and formally
analyze languages in a generic way, but only after the language is already defined. Additionally, as discussed
in Section ?? The following important question remains largely open to the working programming language
designer, and not only:

\[\text{Is there any language definitional framework that, at the same time,}\]

\(^1\)In standalone K definitions, the language syntax is annotated with K attributes when the syntax is defined, not as part of the
semantics as we did in Figure 1.
1. Gives a strong intuition, even precise recipes, on how to define a language?

2. Same for language-related definitions, such as type checkers, type inferences, abstract interpreters, safety policy or domain-specific checkers, etc.?

3. Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?

4. Is modular, that is, adding new features to a language does not require modifying existing definitions of unrelated features? Modularity is crucial for scalability and reuse.

5. Is generic, that is, not tied to any particular programming language or paradigm?

6. Supports naturally non-determinism and concurrency?

7. Is executable, so one can test language or formal analyzer definitions, as if one already had an interpreter or a compiler for one’s language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.

8. Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one’s language is non-deterministic or/and concurrent?

9. Has a formal semantics, so one can also carry out proofs about programs? A fix-point, or an initial model, semantics is necessary if proofs by induction are desired.

We believe that the list above contains a minimal set of desirable features that an ideal language definitional framework should have. Unfortunately, the current practice is to take the above desired features one at a time, temporarily or permanently declaring the others as something else. We next list how, in our view, current practices and language definitional styles fail to satisfy the above-mentioned requirements.

1. To formally capture one’s intuition about a language feature to be defined, one may use a big-step or a small-step SOS reduction semantics, with or without evaluation contexts, typically on paper, without any machine support. Sometimes this so-called “formal” process is pushed to extreme in what regards its informality, in the sense that one can see definitions of some language features using one definitional style and of other features using another definitional style, without ever proving that the two definitional styles can co-exist in the claimed form for the particular language under consideration. For example, one may use a big-step SOS to give semantics to a code-self-generation extension of Java, while using a small-step SOS to define the concurrency semantics of Java. However, once one has concurrency and shared memory, one cannot have a big-step SOS definition of almost anything. An ideal language definitional framework should provide a uniform, compact and rigorous way to modularly define various language features, avoiding the need to define different language features following different styles.

2. To define a type system or a (domain-specific or not) safety policy for a language, one may follow a big-step-like definitional style, or even simply provide an algorithm to serve as a formal definition. While this appears to be, and in many cases indeed is acceptable, there can be a significant formal gap between the actual language semantic definition and its type system or safety policy regarded as mathematical objects, because in order to carry out proofs relating the two one needs one common formal ground. In practice, one typically ends up encoding the two in yet another framework, claimed to be “richer”, and then carry out the proofs within that framework. But how can one make sure that the encodings are correct? Do they serve as alternative definitions for that sole purpose? An ideal definitional framework should have all the benefits of the richer framework, at no additional notational
or logical complexity, yet capturing the complete meaning of the defined constructs. In other words, in an ideal framework one should define a language as a mathematical object, say $L$, and a type system or other abstract interpretation of it as another mathematical object over the same formalism, say $L'$, and then carry out proofs relating $L$ and $L'$ using the provided proof system of the definitional framework. $L$, $L'$, as well as other related definitions, should be readable and easy to understand enough so that one does not feel the drive to give alternative, more intuitive definitions using a more informal notation or framework.

3. Some popular language definitional frameworks are incapable of defining even existing language features. In our view, the fact that a particular language feature is supported in some existing language serves as the strongest argument that that feature may be desirable, so an ideal language definitional framework must simply support it; in other words, one cannot argue against the usefulness of that feature just because one’s favorite definitional framework does not support it. For example, since in standard SOS definitions (not including reduction semantics with evaluation contexts) the control flow information of a program is captured within the structure of the proof derivation, and since proof derivations are not first class objects in these formalisms, it makes it very hard, virtually almost impossible in these formalisms to define complex control intensive language constructs like, e.g., call-with-current-continuation (callcc). Another important example showing that conventional definitional frameworks fail to properly support existing common language features is concurrency: most frameworks enforce an interleaving semantics, which may not necessarily always be desired. Concurrency is further discussed in item 5 below. Some frameworks provide a “well-chosen” set of constructs, shown to be theoretically sufficient to define any computable function or algorithm, and then propose encodings of other language features into the set of basic ones; examples in this category are Turing machines or the plethora of (typed or untyped) λ-calculi, or π-calculi, etc. While these basic constructs yield interesting and meaningful idealized programming languages, using them to encode other language features is, in our view, inappropriate. Indeed, encodings hide the intended computational granularity of the defined language constructs; for example, a variable lookup intended to be a one-step operation in one’s language should take precisely one step in an ideal framework (not hundreds/thousands of steps as in a Turing machine or lambda calculus encoding, not even two steps: first get location, then get value).

4. As Mosses pointed out in [53], SOS is non-modular. Even in the original notes on SOS [60][61], Plotkin had to modify the definition of simple arithmetic expressions three times as his initial language evolved [53]. Even more frustratingly, to add an innocent abrupt termination statement to a language defined using SOS, say a halt, one would need to more than double the total number of rules: each language construct needs to be allowed to propagate the halting signal potentially generated by its arguments. Also, as one needs to add more items into configurations to support new language features, in SOS one needs to change again every rule to include the new items; note that there are no less than $7 + n \times 10$ configuration items, where $n$ is the number of threads, in the configuration of KOOL (which is a comparatively simple language) as shown in Figure CHANGETHIS. It can easily become very annoying and error prone to modify a large portion of unrelated existing definitions when adding a new feature. A language designer may be unwilling to add a new feature or improve the definition of an existing one, just because of the large number of required changes. Informal writing conventions are sometimes adopted to circumvent the non-modularity of SOS. For example, in the definition of Standard ML [49], Milner and his collaborators propose a “store convention” to avoid having to mention the store in every rule, and an “exception convention” to avoid having to double the number of rules for
the sole purpose of supporting exceptions. As rightfully noticed by Mosses \cite{53}, such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Mosses’ Modular SOS \cite{53} (MSOS) brings modularity to SOS in a formal and elegant way, by grouping the non-syntactic configuration items into transition labels, and allowing rules to mention only those items of interest from each label. As discussed in Section 3.6, MSOS still inherits all the remaining limitations of SOS.

5. A non-generic framework, i.e., one building upon a particular programming language or paradigm, may be hard or impossible to use at its full strength when defining a language that crosses the boundaries of the underlying language or paradigm. For example, a framework enforcing object or thread communication via explicit send and receive messages may require artificial encodings of languages that opt for a different communication approach (e.g., shared memory), while a framework enforcing static typing of programs in the defined language may be inconvenient for defining dynamically typed or untyped languages. In general, a framework providing and enforcing particular ways to define certain types of language features would lack genericity. Within an ideal framework, one can and should develop and adopt methodologies for defining certain types of languages or language features, but these should not be enforced. This genericity requirement is derived from the observation that today’s programming languages are so diverse and based on orthogonal, sometimes even conflicting paradigms, that, regardless of how much we believe in the superiority of a particular language paradigm, be it object-oriented, functional or logical, a commitment to any existing paradigm would significantly diminish the strengths of a language definitional framework.

6. By inherently enforcing an interleaving semantics for concurrency, existing reduction semantics definitions (including ones based on evaluation contexts) can only capture a projection of concurrency (when one’s goal is to define a truly concurrent language), namely its resulting non-determinism. Proponents of existing reduction semantics approaches may argue that the resulting non-deterministic behavior of a concurrent system is all that matters, while proponents of true concurrency may argue that a framework which does not support naturally concurrent actions, i.e., actions that take place at the same time, is not a framework for concurrency. We do not intend to discuss the important and debatable distinctions between non-determinism and interleaving vs. true concurrency here. The fact that there are language designers who desire an interleaving semantics while others who desire a true concurrency semantics for their language is strong evidence that an ideal language definitional framework should simply support both, preferably with no additional settings of the framework, but rather via particular definitional methodologies within the framework.

7. Most existing language definitional frameworks are, or until relatively recently were, lacking tool support for executability. Without the capability to execute language definitions, it is virtually impossible to debug or develop large and complex language definitions in a reasonable period of time. The common practice today is still to have a paper definition of a language using one’s desired formalism, and then to implement an interpreter for the defined language following in principle the paper definition. This approach, besides the inconvenience of having to define the language twice, guarantees little to nothing about the appropriateness of the formal, paper definition. Compare this approach to an approach where there is no gap between the formal definition and its implementation as an interpreter. While any definition is by definition correct, one gets significantly more confidence in the appropriateness of a language definition, and is less reluctant to change it, when one is able to run it as is on tens or hundreds of programs. Recently, executability engines have been proposed both for MSOS (the MSOS tool, implemented by Braga and collaborators in Maude \cite{16}) and for reduction semantics with evaluation contexts (the PLT Redex tool, implemented by Findler and his collaborators in Scheme \cite{37}).
A framework providing efficient support for executability of formal language definitions may eliminate entirely the need to implement interpreters, or type checkers or type inferencers, for a language, because one can use directly the formal definition for that purpose.

8. While executability of language definitions is indispensable when designing non-trivial languages, one needs richer tool support when the language is concurrent. Indeed, it may be that one’s definition is appropriate for particular thread or process interleavings (e.g., when blocks are executed atomically), but that it has unexpected behaviors for other interleavings. Moreover, somewhat related to the desired computational granularity of language constructs mentioned in item 3 above, one may wish to exhaustively investigate all possible interleavings or executions of a particular concurrent program, to make sure that no undesired behaviors are present and no desired behaviors are excluded. Moreover, when the state space of the analyzed program is large, manual analysis of behaviors may not be feasible; therefore, model-checking and/or safety property analysis are also desirable as intrinsic components of an ideal language definitional framework.

9. To prove properties about programs in a defined programming language, or properties about the programming language itself, as also mentioned in item 2 above, the current practice is to encode/redefine the language semantics in a “richer” framework, such as a theorem prover, and then carry out the desired proofs there. Redefining the semantics of a fully fledged programming language in a different formalism is a highly nontrivial, error prone and tedious task, possibly taking months; automated translations may be possible when the original definition of the language is itself formal, though one would need to validate the translator. In addition to the formal gap mentioned in item 2 due to the translation itself, this process of redefining the language is simply inconvenient. An ideal language definitional framework should allow one to have, for each language, one definition serving all purposes, including all those mentioned above. Most likely, proofs about programs or programming languages will need induction. In order for induction to be a valid proof principle, either it needs to be hardwired as one or more proof rules in the framework when the framework is proof-theoretical, or the framework must possess a fixed-point or an initial model semantics when it is model-theoretical.

Peter: reuse; Chucky: proof modularity

There are additional desirable, yet of a more subjective nature and thus harder to quantify, requirements of an ideal language definitional framework. For example, it should be simple and easy to understand, teach and use by mainstream enthusiastic language designers, not only by language experts—in particular, an ideal framework should not require its users to have advanced concepts of category theory, logics, or type theory, in order to use it. Also, it should have good data representation capabilities and should allow proofs of theorems about programming languages that are easy to comprehend. Additionally, a framework providing support for parsing programs directly in the desired language syntax may be desirable to one requiring the implementation of an additional, external to the definitional setting, parser.

The nine requirements above are nevertheless ambitious. Some proponents of existing language definitional frameworks may argue that their favorite framework has these properties; however, a careful analysis of existing language definitional frameworks reveals that they actually fail to satisfy some, sometimes many, of these ideal features (we discuss several such frameworks and their limitations in Chapter 3). Others may argue that their favorite framework has some of the properties above, the “important ones”, declaring the other properties either “not interesting” or “something else”. For example, one may say that what is important in one’s framework is to get a dynamic semantics of a language, but its (model-theoretical)
algebraic denotational semantics, proving properties about programs, model checking, etc., are “something else” and therefore are allowed to need a different “encoding” of the language. Our position is that an ideal language definitional framework should not compromise any of the nine requirements above. Whether K satisfies all the requirements above or not is, and probably should always be, open. What we can mention with regards to this aspect, though, is that K was motivated and stimulated by the observation that the existing language definitional frameworks fail to fully satisfy these minimal requirements; consequently, K’s design and development was conducted aiming explicitly to fulfill all nine requirements discussed above.

K’s actual merit is in addressing the first four requirements in our list above, namely it proposes a scalable, modular and comprehensive formal approach to define programming languages and language-related features and analyses. While the first four requirements are specific to programming languages, the remaining five, namely genericity, support for concurrency, executability, formal analysis and formal semantics, transcend the domain of programming languages. Indeed, the latter are desired features of system specification formalisms in general, an area of great interest to many researchers and, consequently, with a variety of approaches, methods, techniques and supporting tools. K’s current approach is to avoid proposing a new general purpose system specification formalism, with a new semantics and with tool support developed from scratch (in other words, to avoid “reinventing the wheel”), but instead to build upon the vast existing theoretical and tool infrastructure of rewrite logic [42][17]. This decision more than paid off. For example, we can carry out proofs about language definitions and corresponding type systems using the existing proof machinery of rewrite logic (see Section CHANGETHIS). Also, our definition of a polymorphic type inferencer in K using Maude (see Section CHANGETHIS) significantly outperforms the current type inferencer of SML and slightly that of Haskell, while slightly under-performs that of OCAML. Experiments performed with the K definition of Java 1.4 using Maude’s builtin model checker, that translates into a model-checker for Java for free, outperforms state-of-the art model checkers developed specifically for Java, such as Java PathFinder [23].

In this report, syntax is defined using only conventional and familiar context-free grammars, extended with lists and sets in a natural way. K is formally defined avoiding the algebraic notation, but its intuitions are borrowed from rewrite logic, an inherently algebraic meta-framework.

We generate all these structural equations automatically from strictness attributes in our implementation (see Section ??).

structure of the paper, machine data (fsl), what versions of maude, ocaml, haskell, sml

have a section on what this book is not about:
- advanced parsing; we assume advanced parsing is done outside and an AST is obtained; nevertheless, we still allow human readable, mixfix syntactic notation, etc.

say that concurrency is nothing special, all defined languages are concurrent

14
Chapter 2

Background and Preliminaries
say that we use tt font for code, including integers and program
variables, and that we use italic or mathematical font for meta-
variables ranging over the various syntactic categories.

add Computability, Degrees ($\Pi^0_2$, etc.)

add Initial algebra, induction, consistency, $\Pi^0_2$ results, etc.

say that the various mathematical domains defined here will be used
as a basis for the various language semantics in Chapter 3

powerset of a set

Equational logic, or the logic of equality, is one of the most basic logical formalisms. Term rewriting is a
standard computational model supported by many systems and languages. rewrite logic, not to be confused
with term rewriting, organizes term rewriting modulo equations as a logic with a complete proof system and
initial model semantics. Maude is an efficient rewrite logic engine.

2.1 Mathematical Background

lists, sets, multisets, $|A|$, $\infty$, $0^\omega$, $0^*$

Lists, sets, multisets, and functions (partial or total) are basic mathematical structures that are indis-
pensable when defining programming language semantics. We assume the reader familiar with elementary
mathematical notions, the role of this section being to introduce our notational conventions.

2.1.1 Functions

Discuss Currying; needed in Section 4.5

A function, or map, uniquely relates elements in its source set, also called its domain, to elements in its
target set, also called its co-domain.

**Definition 1.** [$A \rightarrow B$] is the set of **functions**, or **maps**, defined from $A$ to $B$. To state that $f$ is a function from
$A$ to $B$, we may write either $f \in [A \rightarrow B]$ or the more common $f : A \rightarrow B$. If $f : A \rightarrow B$ then $A$ is called the
domain or the source of $f$ and $B$ is called the co-domain or the target of $f$. If $a \in A$ then $f(a) \in B$ denotes
the unique element in $B$ that $f$ relates to $a$.

Unlike partial functions which are discussed below, functions are defined on all the elements in their
source, reason for which we also call their source domain.
**Definition 2.** If \( f: A \to B \), \( a \in A \) and \( b \in B \), then we let \( f[b/a] : A \to B \) denote the function that **updates** \( f \) in \( a \) to \( b \), defined formally as follows:

\[
\begin{array}{ll}
  f(b/a)(a') = \begin{cases} 
    f(a') & \text{when } a' \neq a, \\
    b & \text{when } a' = a.
  \end{cases}
\end{array}
\]

**Exercise 1.** With the notation in Definition 2, could it be the case that \( f[b/a] = f \)? If \( a, a' \in A \) and \( b, b' \in B \), is it always the case that \( f[b/a][b'/a'] = f[b'/a'][b/a] \)? What if \( b = b' \)?

**Exercise 2.** Show that if \( A \) is empty then the set \( [A \to B] \) contains precisely one function, regardless of whether \( B \) is empty or not.

**Definition 3.** By abuse of notation, we let \( \emptyset \) also denote the unique function in \( [\emptyset \to B] \).

**Exercise 3.** Show that \( [A \to B] \) contains only one function if and only if \( A = \emptyset \) or \( |B| = 1 \).

**Exercise 4.** Show that the set \( [A \to \emptyset] \) is empty when \( A \neq \emptyset \).

**Exercise 5.** Show that if \( A \) and \( B \) are finite, then \( [A \to B] \) is also finite and \( |[A \to B]| = |B|^{|A|} \).

### 2.1.2 Partial Functions

A partial function, or partial map, uniquely relates a subset of elements in its source set, called its domain, to elements in its target set.

**Definition 4.** \( [A \to B] \) is the set of **partial functions**, or **partial maps**, from \( A \) to \( B \). We may write either \( f \in [A \to B] \) or \( f: A \to B \) when stating that \( f \) is a partial function from \( A \) to \( B \). If \( f: A \to B \) then \( A \) is the **source** of \( f \) and \( B \) is the **target** of \( f \). Like for functions, if \( a \in A \) then we let \( f(a) \in B \) be the unique element in \( B \) that \( f \) relates to \( a \), in case such an element exists; if such an \( f(a) \in B \) does not exists for a given \( a \in A \), then we say that \( f \) is **undefined** in \( a \), or that \( f(a) \) is **undefined**. \( \text{Dom}(f) \overset{\text{def}}{=} \{ a \in A \mid f(a) \text{ is defined} \} \) is the **definitional domain**, or just the **domain**, of \( f \). If \( \text{Dom}(f) = A \) then \( f \) is a function like in Definition 1 and we may call it a **total** (partial) function. If \( \text{Dom}(f) \) is finite then \( f \) is called a **partial finite-domain function**. We let \( [A \to B]_{\text{finite}} \) denote the set of partial finite-domain functions in \( [A \to B] \).

Therefore, ordinary functions are special partial functions, namely ones defined on all the elements in their sources. The function update operation in Definition 2 can be naturally extended to partial functions:

**Definition 5.** If \( f: A \to B \), \( a \in A \) and \( b \in B \), then we let \( f[b/a] : A \to B \) denote the partial function that **updates** \( f \) in \( a \) to \( b \), defined formally as follows:

\[
\begin{array}{ll}
  f(b/a)(a') = \begin{cases} 
    f(a') & \text{when } a' \in \text{Dom}(f) \text{ and } a' \neq a, \\
    b & \text{when } a' = a, \\
    \text{undefined} & \text{when } a' \notin \text{Dom}(f) \cup \{a\}
  \end{cases}
\end{array}
\]
If one assumes that \( f(a') \) can either be an element in \( B \) or the special element \textit{undefined}, the definition above can be stated more compactly, like Definition 2:

\[
f[b/a](a') = \begin{cases} f(a') & \text{when } a' \neq a, \\ b & \text{when } a' = a. \end{cases}
\]

**Exercise 6.** With the notation in Definition 5 show that \( \text{Dom}(f[b/a]) = \text{Dom}(f) \cup \{a\} \). Under what conditions \( f[b/a] = f \)?

**Exercise 7.** Show that \( \emptyset 
arrow B = \emptyset 
rightarrow B \) for any set \( B \).

The exercise above says, in particular, that the function \( \emptyset : \emptyset \rightarrow B \) is total.

**Definition 6.** For any \( A \) and \( B \), we let \( \perp : A \rightarrow B \) denote the \textit{undefined everywhere partial function} from \( A \) to \( B \), that is, the partial function with \( \perp(a) \) undefined for any \( a \in A \).

Therefore, unlike \( [A \rightarrow B] \), the set \( [A \rightarrow B] \) always contains at least one element, the undefined everywhere partial function \( \perp : A \rightarrow B \).

**Exercise 8.** Show that the partial functions \( \emptyset : \emptyset \rightarrow B \) and \( \perp : \emptyset \rightarrow B \) are equal.

**Exercise 9.** Show that \( [A \rightarrow B] \) contains only one element if and only if \( A = \emptyset \) or \( B = \emptyset \) or \( |B| = 1 \).

**Exercise 10.** If \( A \) and \( B \) are finite then \( [A \rightarrow B] \) is also finite; if that is the case, can you express \( |[A \rightarrow B]| \) as a closed expression in terms of \( |A| \) and \( |B| \)?

### 2.1.3 Context-Free Grammars and the Mixfix Algebraic Notation

BNF (Bachus-Naur Form) or CFG (Context Free Grammar) notations to describe the syntax of programming languages are very common. We also used BNF to define the syntax of our simple language in the lectures on SOS. From here on, we can interchangeably use the mix-fix notation instead of BNF or CFG, so it is important to understand the relationship between the mix-fix notation and BNF and CFG:

**Exercise 11.** Argue that the mix-fix notation is equivalent to BNF and CFG. Find an interesting simple example language and express its syntax using the mix-fix notation, BNF, and production-based CFG (e.g., \( A \rightarrow AB \mid a \) and \( B \rightarrow BA \mid b \)).

---

we may write \( a \in \text{AExp} \) instead of \( a \in \mathcal{L}(\text{AExp}) \)

---

introduce the notations

---

introduce “algebraic CFG”: extend CFG with these.

---

say that we allow the use of parentheses for disambiguation.

---

the following is also something to take care of:
e.g., \texttt{List\{S\}} defines comma-separated lists of elements of type \(S\), and could be expressed with the CFG rule \(\text{List}\{S\} ::= \cdot \mid S (, S^*)\), with the additional understanding that ‘\(,\)’ is associative and \(\cdot\) satisfies the unit axioms.

say that from a parsing perspective sets and multisets are no different than lists; also, say that in RWL set/multiset axioms are important since rewriting happens modulo those, while \(K\) has builtin support for these; in \(K\) rules adapt modulo their axioms and not the world.

say that if \(t\) is a term using the CFG notation, we usually let \(\bar{t}\) denote the algebraic version of \(t\); in case \(t\) has any “parameter”, as needed for example in deduction rules, then that parameter becomes of variable in \(\bar{t}\) of appropriate sort. See, for example, the translations of the various semantic approaches to RL and their correctness proofs for what we need to cover here.

More precisely, one sort is added for each syntactic category, or non-terminal in the grammar, and one operation \(\gamma_\cdot : NT(\gamma) \to S\) is added for each production \(S ::= \gamma\) in the grammar, where \(\gamma_\cdot\) replaces each non-terminal in \(\gamma\) by an underscore and \(NT(\gamma)\) is the product of the sorts corresponding to the non-terminals in \(\gamma\), in the order of their appearance. For example, the production \(\text{Stmt} ::= \text{if } BExp \text{ then Stmt else Stmt}\) is associated the algebraic operation

\[
\text{if.then.else_.} : BExp \times \text{Stmt} \times \text{Stmt} \to \text{Stmt}.
\]

Applying this syntax algebrization technique to the entire syntax of IMP in Figure 3.1, we obtain the algebraic signature in Figure 3.2.

\section*{Notes}
- mixfix notation as equivalent to CFG
- BNF is an acronym for ”Backus Naur Form”. John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language (This was for the description of the ALGOL 60 programming language, see [Naur 60]). To be precise, most of BNF was introduced by Backus in a report presented at an earlier UNESCO conference on ALGOL 58. Few read the report, but when Peter Naur read it he was surprised at some of the differences he found between his and Backus’s interpretation of ALGOL 58. He decided that for the successor to ALGOL, all participants of the first design had come to recognize some weaknesses, should be given in a similar form so that all participants should be aware of what they were agreeing to. He made a few modifications that are almost universally used and drew up on his own the BNF for ALGOL 60 at the meeting where it was designed. Depending on how you attribute presenting it to the world, it was either by Backus in 59 or Naur in 60. (For more details on this period of programming languages history, see the introduction to Backus’s Turing award article in Communications of the ACM, Vol. 21, No. 8, August 1978. This note was suggested by William B. Clodius from Los Alamos Natl. Lab).

\subsection*{2.1.4 Basic Term Rewriting}

\begin{itemize}
\item define confluence
\end{itemize}
A term rewrite system (TRS) consists of a set of uninterpreted operations, possibly over different syntactic categories, or sorts, and a set of rewrite rules of the form $l \rightarrow r$, where $l$ and $r$ are terms possibly containing variables with possibly multiple occurrences. Term rewriting is a method of computation that works by progressively changing (rewriting) a term, using a TRS. A rule can apply to the entire term being rewritten or to a subterm of the term. First, a match within the current term is found. This is done by finding a substitution, $\theta$, from variables to terms such that the left-hand-side of the rule, $l$, matches part or all of the current term when the variables in $l$ are replaced according to the substitution. The matched subterm is then replaced by the result of applying the substitution to the right-hand-side of the rule, $r$. Thus, the part of the current term matching $\theta(l)$ is replaced by $\theta(r)$. The rewriting process continues as long as it is possible to find a subterm, rule, and substitution such that $\theta(l)$ matches the subterm. When no matching subterms are found, the rewriting process terminates, with the final term, a normal form, being the result of the computation.

One of the distinctive features of term rewriting is that rules apply in a context insensitive manner; in other words, the larger context in which a rule is matched does not influence the applicability of the rewrite rule. Rewriting, like other methods of computation, can continue forever. The following is a rewrite definition of a factorial language construct in a hypothetical calculator language:

$$
\begin{align*}
\text{Exp} &::= \text{Nat} \mid \text{Exp} * \text{Exp} \mid \text{Exp} - \text{Exp} \mid \ldots \mid \text{Exp}! \\
0! &\rightarrow 1 \\
n! &\rightarrow n * (n - 1)!, \quad \text{when } n \geq 1
\end{align*}
$$

There are a large number of term rewriting engines, including ASF\textsuperscript{[82]}, Elan\textsuperscript{[12]}, Maude\textsuperscript{[17]}, OBJ\textsuperscript{[28]}, Stratego\textsuperscript{[85]}, and others, some of which are capable of executing several million rewrites per second. Rewriting is also a fundamental part of many existing functional languages and theorem provers. Term rewriting and rewrite logic definitions of languages or language features, if executed on efficient rewrite engines, can be as efficient or even outperform implementations of interpreters or other tools in conventional programming languages. This point can be quickly reflected even on the simple calculator language above. For example, in our experiments executing the definition above, the factorial of 50,000, a number of 213,237 digits, was calculated in 18.620 seconds by Maude and in 19.280 and 16.770 seconds by the programming languages ML and Scheme, respectively. While for this particular example one may say that the quality of large integers libraries plays the major role, we obtained similar results for many other experiments.
proof systems, side conditions of proof rules (make text below make sense)

The conditions which make a rule applicable, such as, for example, “where i is the sum of \(i_1\) and \(i_2\)” in rule (Add), are called side conditions. Side conditions typically constrain variables over recursively enumerable domains (e.g., \(i, i_1\) and \(i_2\) range over integer numbers in rule (Add)). Therefore, each SOS rule comprises a recursively enumerable collection of concrete instance rules. One could also move the side-conditions into the actual condition of the rule (above the line), but many rules that are unconditional would then become conditional and thus harder to type and read.

add background here in order for the next to make sense

An SOS definition is a collection of parametric rules of the form:

\[
\begin{array}{c}
\text{transition}_1, \text{transition}_2, \ldots, \text{transition}_k \\
\hline
\text{transition}
\end{array}
\]

The transitions above the line are called the condition of the rule, and the transition below the line is called the conclusion of the rule. The intuition here is that \(\text{transition}\) is possible whenever \(\text{transition}_1, \text{transition}_2, \ldots, \text{transition}_k\) are possible. We may also say that \(\text{transition}\) is derivable, or can be inferred, from \(\text{transition}_1, \text{transition}_2, \ldots, \text{transition}_k\). This reflects the fact that an SOS definition can (and should) be viewed as a logical system, where one can deduce possible behaviors of programs.

Structural induction on proof/derivation trees

For Section 2.3.

introduce the notation \(\Gamma \vdash \varphi\)

axioms vs. proof rules; former a special case of the latter

axiom / proof rule schemata

\(\vdash \varphi\) is a shorthand for \(\emptyset \vdash \varphi\); \(\varphi\) called a theorem in the corresponding logic/proof system.

general properties about entailment, such as \(\vdash \varphi\) implies \(\Gamma \vdash \varphi\), etc. (obviously, in a more general form).

Proposition 1. If \(\Gamma \vdash \varphi\) then there is some finite subset \(\Gamma_0\) of \(\Gamma\) such that \(\Gamma_0 \vdash \varphi\).
2.2 Basic Computability Elements

In this section we recall very basic concepts of computability theory needed for other results in the book, and introduce our notation for these. This section is by no means intended to serve as a substitute for much thorough presentations found in dedicated computability textbooks (some mentioned in Section 2.2.4).

2.2.1 Turing Machines

Turing machines are abstract computational models used to formally capture the informal notion of a computing system. The Church-Turing thesis postulates that any computing system, any algorithm, or any program in any programming language running on any computer, can be equivalently simulated by a Turing machine. Having a formal definition of computability allows us to rigorously investigate and understand what can and what cannot be done using computing devices, regardless of what languages are used to program them. Intuitively, by a computing device we understand a piece of machinery that carries out tasks by successively applying sequences of instructions and using, in principle, unlimited memory; such sequences of instructions are today called programs, or procedures, or algorithms.

Turing machines are used for their theoretical value; they are not meant to be physically built. A Turing machine is a finite state device with infinite memory. The memory is very primitively organized, as one or more infinite tapes of cells that are sequentially accessible through heads that can move to the left or to the right cell only. Each cell can hold a bounded piece of data, typically a Boolean, or bit, value. The tape is also used as the input/output of the machine. The computational steps carried out by a Turing machine are also very primitive: in a state, depending on the value in the current cell, a Turing machine can only rewrite the current cell on the tape and/or move the head to the left or to the right. Therefore, a Turing machine does not have the direct capability to perform random memory access, but it can be shown that it can simulate it.

There are many equivalent definitions of Turing machines in the literature. We prefer one with a tape that is infinite at both ends and describe it next (interestingly, an almost identical machine was proposed by Emil Post independently from Alan Turing also in 1936; see Section 2.2.4). Consider a mechanical device which has associated with it a tape of infinite length in both directions, partitioned in spaces of equal size, called cells, which are able to hold either a 0 or an 1 and are rewritable. The device examines exactly one cell at any time, and can, potentially nondeterministically, perform any of the following four operations (or commands):

1. Write a 1 in the current cell;
2. Write a 0 in the current cell;
3. Shift one cell to the right;
4. Shift one cell to the left.

The device performs one operation per unit time, called a step. We next give a formal definition.

Definition 7. A (deterministic) Turing machine $M$ is a 6-tuple $(Q, B, q_s, q_h, C, M)$, where:

- $Q$ is a finite set of internal states;
- $q_s \in Q$ is the starting state of $M$;
- $q_h \in Q$ is the halting state of $M$;
- $B$ is the set of symbols of $M$; we assume without loss of generality that $B = \{0, 1\}$;
- $C = B \cup \{\rightarrow, \leftarrow\}$ is the set of commands of $M$;
- $M : (Q - \{q_h\}) \times B \rightarrow Q \times C$ is a total function, the transition function of $M$.

We assume that the tape contains only 0’s before the machine starts performing.
Our definition above is for deterministic Turing machines. One can also define nondeterministic Turing machines by changing the transition function \( M \) into a relation. Non-deterministic Turing machines have the same computational power as the deterministic Turing machines (i.e., they compute/decide the same classes of problems; computational speed or size of the machine is not a concern here), so, for our purpose in this section, we limit ourselves to deterministic machines.

A configuration of a Turing machine is a 4-tuple consisting of an internal state, a current cell, and two infinite strings (notice that the two infinite strings contain only 0’s starting with a certain cell), standing for the cells on the left and for the cells on the right of the current cell, respectively. We let \( (q, LbR) \) denote the configuration in which the machine is in state \( q \), with current cell \( b \), left tape \( L \) and right tape \( R \). For convenience, we write the left tape \( L \) backwards, that is, its head is at its right end; for example, \( Lb \) appends a \( b \) to the left tape \( L \). Given a configuration \( (q, LbR) \), the content of the tape is \( LbR \), which is infinite at both ends. We also let \( (q, LbR) \rightarrow M (q', L' b' R') \) denote a configuration transition under one of the four commands. A set \( S \subseteq B^* \) is recursively enumerable (r.e.,) or semi-decidable, respectively co-recursively enumerable (co-r.e.) or co-semi-decidable, iff there is some Turing machine \( M \) which terminates on precisely all inputs \( b_1b_2\cdots b_n \in S \), respectively on precisely all inputs \( b_1b_2\cdots b_n \notin S \), and is recursive or decidable iff it is both r.e. and co-r.e.

Note that a Turing machine as we defined it cannot get stuck in any state but \( q_h \), because the mapping \( M \) is defined everywhere except in \( q_h \). Therefore, for any given input, a Turing machine carries out a determined succession of steps, which may or may not terminate. A Turing machine can be regarded as an idealized, low-level programming language, which can be used for computations by placing a certain input on the tape and letting it run; if it terminates, the result of the computation can be found on the tape. Since our Turing machines have only symbols 0 and 1, one has to use them ingeniously to encode more complex inputs, such as natural numbers. There are many different ways to do this. A simple tape representation of natural numbers is to represent a number \( n \) by \( n \) consecutive cells containing the bit 1. This works, however, only when \( n \) is strictly larger than 0. Another representation, which also accommodates \( n = 0 \), is as a sequence of \( n + 1 \) cells containing 1. With this latter representation, one can then define Turing machines corresponding
**States and transition function (graphical representation to the right):**

\[
Q = \{ q_s, q_h, q_1, q_2 \}
\]

\[
M(q_s, 0) = (q_1, \rightarrow)
\]

\[
M(q_s, 1) = \text{anything}
\]

\[
M(q_1, 0) = (q_2, 1)
\]

\[
M(q_1, 1) = (q_1, \rightarrow)
\]

\[
M(q_2, 0) = (q_h, 0)
\]

\[
M(q_2, 1) = (q_2, \leftarrow)
\]

**Sample computation:**

\[
(q_s, 0^m1110^o) \rightarrow_M (q_1, 0^m1110^o) \rightarrow_M (q_1, 0^m1110^o) \rightarrow_M (q_1, 0^m1110^o) \rightarrow_M
\]

\[
(q_1, 0^m1110^o) \rightarrow_M (q_2, 0^m1110^o) \rightarrow_M (q_2, 0^m1110^o) \rightarrow_M (q_2, 0^m1110^o) \rightarrow_M
\]

\[
(q_2, 0^m1110^o) \rightarrow_M (q_2, 0^m1110^o) \rightarrow_M (q_h, 0^m1110^o)
\]

Figure 2.1: Turing machine \( M \) computing the successor function, and sample computation

to functions that take any natural numbers as input and produce any natural numbers as output. For example, Figure 2.1 shows a Turing machine computing the successor function. Cells containing 0 can then be used as number separators, when more natural numbers are needed. For example, a Turing machine computing a binary operation on natural numbers would run on configurations \((q_s, 0^m1^{n+1}01^{k+1}1^o)\) and would halt on configurations \((q_h, 0^n01^{k+1}1^o)\), where \(m, n, k\) are natural numbers.

One can similarly have tape encodings of rational numbers; for example, one can encode the number \(m/n\) as \(m\) followed by \(n\) with two 0 cells in-between (and keep the one-0-cell-convention for argument separation). Real numbers are not obviously representable, though. A Turing machine is said to compute a real number \(r\) iff it can finitely approximate \(r\) (for example using a rational number) with any desired precision; one way to formalize this is as follows: Turing machine \(M_r\) computes \(r\) iff when run on input natural number \(p\), it halts with result rational number \(m/n\) such that \(|r - m/n| < 1/10^p\). If a real number can be computed by a Turing machine then it is called Turing computable. Many real numbers, e.g., \(\pi\), \(e\), \(\sqrt{2}\), etc., are Turing computable.

### 2.2.2 Universal Machines, the Halting Problem, and Decision Problems

Since Turing machines have finite descriptions, they can be encoded themselves as natural numbers. Therefore, we can refer to “the \(k^{\text{th}}\) Turing machine”, where \(k\) is a natural number, the same way we can refer to the \(i^{\text{th}}\) input to a Turing machine. A universal Turing machine is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input thereof from its own tape. There are various constructions of universal Turing machines in the literature, which we do not repeat here. We only notice that we can construct such a universal machine \(\mathcal{U}\) which terminates precisely on all inputs of the form \(1^i01^i\) where Turing machine \(k\) terminates on input \(i\). This immediately implies that the language \(\{1^i01^i \mid \text{Turing machine } k \text{ terminates on input } i\}\) is recursively enumerable. However, the undecidability of the famous halting problem (does a given Turing machine terminates on a given input?) implies that this language is not recursive; more specifically, it is not co-recursively enumerable.

Since the elements of many mathematical domains can be encoded as words in \(B^*\), the terminology in Definition 8 is also used for decision problems over such domains. For example, the decision problem of
whether a graph given as input has a cycle or not is recursive; in other words, the set of cyclic graphs (under some appropriate encoding in \(B^*\)) is recursive/decidable. Since there is a bijective correspondence between elements in \(B^*\) and natural numbers, and also between tuples of natural numbers and natural numbers, decision problems are often regarded as one- or multi-argument relations or predicates over natural numbers. For example, a subset \(R \subseteq B^*\) can be regarded as a predicate, say of three natural number arguments, where \(R(i, j, k)\) for \(i, j, k \in \text{Nat}\) indicates that the encoding of \((i, j, k)\) belongs to \(R\).

While the halting problem is typically an excellent vehicle to formally state and prove that certain problems are undecidable, so they cannot be solved by computers no matter how powerful they are or what programming languages are used, its reflective nature makes the halting problem sometimes hard to use in practice. The Post correspondence problem (PCP) is another canonical undecidable problem, which is sometimes easier to use to show that other problems are undecidable. The PCP can be stated as follows: given a set of (domino-style) tiles each containing a top and a bottom string of 0/1 bits, is it possible to find a sequence of possibly repeating such tiles so that the concatenated top strings equal the concatenated bottom strings? For example, if the given tiles are the following:

\[
\begin{align*}
1 : & \begin{array}{c}
\circ \\
\bullet \\
\bullet \bullet \\
\end{array} \\
2 : & \begin{array}{c}
\bullet \\
\circ \circ \\
\end{array} \\
3 : & \begin{array}{c}
\bullet \bullet \\
\bullet \\
\end{array}
\end{align*}
\]

then the answer to the PCP problem is positive, because the sequence of tiles 3 2 3 1 yields the same concatenated strings at the top and at the bottom.

### 2.2.3 The Arithmetic Hierarchy

The **arithmetical hierarchy** defines classes of problems of increasing difficulty, called **degrees**, as follows:

\[
\begin{align*}
\Sigma^0_n &= \Pi^0_n = \{ R \mid R \text{ recursive} \} \\
\Sigma^0_0 &= \{ P \mid \exists Q \in \Pi^0_n, \forall i (P(i) \leftrightarrow \exists j Q(i, j)) \} \\
\Pi^0_0 &= \{ P \mid \exists Q \in \Sigma^0_n, \forall i (P(i) \leftrightarrow \forall j Q(i, j)) \}
\end{align*}
\]

For example, the \(\Sigma^0_0\) degree consists of the predicates \(P\) over natural numbers for which there is some recursive predicate \(R\) such that for any \(i \in \text{Nat}\), \(P(i)\) holds iff \(R(i, j)\) holds for some \(j \in \text{Nat}\). It can be shown that \(\Sigma^0_0\) contains precisely the recursively enumerable predicates. Similarly, \(\Pi^0_0\) consists of the predicates \(P\) for which there is some recursive predicate \(R\) such that for any \(i \in \text{Nat}\), \(P(i)\) holds iff \(R(i, j)\) holds for all \(j \in \text{Nat}\), which is precisely the set of co-recursively enumerable predicates. An important degree is also \(\Pi^0_2\), which consists of the predicates \(P\) over natural numbers for which there is some recursive predicate \(R\) such that for any \(i \in \text{Nat}\), \(P(i)\) holds iff for any \(j \in \text{Nat}\) there is some \(k \in \text{Nat}\) such that \(R(i, j, k)\) holds. A prototypical \(\Pi^0_2\) problem is the following: giving a Turing machine \(M\), does it terminate on all inputs? The complexity of the problem stays in the fact that there are infinitely many (but enumerable) inputs, so one can never be “done” with testing them; moreover, even for a given input, one does not know when to stop running the machine and reject the input. However, if one is given for each input accepted by the machine a run of the machine, then one can simply check the run and declare the input indeed accepted. Therefore, \(M\) terminates on all inputs iff for any input there exists some accepting run of \(M\), which makes it a \(\Pi^0_2\) problem because checking whether a given run on a given Turing machine with a given input accepts the input is decidable. Moreover, it can be shown that if we pick \(M\) to be the universal Turing machine \(U\) discussed above, then the following problem, which we refer to as **Totality** from here on, is in fact \(\Pi^0_2\)-complete:

\[
\begin{align*}
\Sigma^0_n &= \Pi^0_n = \{ R \mid R \text{ recursive} \} \\
\Sigma^0_0 &= \{ P \mid \exists Q \in \Pi^0_n, \forall i (P(i) \leftrightarrow \exists j Q(i, j)) \} \\
\Pi^0_0 &= \{ P \mid \exists Q \in \Sigma^0_n, \forall i (P(i) \leftrightarrow \forall j Q(i, j)) \}
\end{align*}
\]
**Input**: A natural number \( k \);

**Output**: Does \( \mathcal{U} \) terminate on all inputs \( 1^k 01^i \) for all \( i \geq 0 \)?

For any \( n \), both degrees \( \Sigma_n^0 \) and \( \Pi_n^0 \) are properly included in both the immediately above degrees \( \Sigma_{n+1}^0 \) and \( \Pi_{n+1}^0 \). There are extensions which define degrees \( \Sigma_n^1 \), \( \Pi_n^2 \), etc., but we do not discuss them here.

### 2.2.4 Notes

The abstract computational model that we call the “Turing machine” today was originally called “the computer” when proposed by Alan Turing in 1936-37 \[80\]. In the original article, Turing imagined not a machine, but a person (the computer) who executes a series of deterministic mechanical rules “in a desultory manner”. In his 1948 essay “Intelligent Machinery”, Turing calls the machine he proposed the *logical computing machine*, a name which has not been adopted, everybody preferring to call it the *Turing machine*.

It is insightful to understand the scientific context in which Turing proposed his machine. In the 1930s, there were several approaches attempting to address Hilbert’s tenth question of 1900, the Entscheidungsproblem (the decision problem). Partial answers have been given by Kurt Gödel in 1930 (and published in 1931), under the form of his famous incompleteness theorems, and in 1934, under the form of his recursion theory. Alonzo Church is given the credit for being the first to effectively show that the Entscheidungsproblem was indeed undecidable, introducing also \( \lambda \)-calculus (discussed in Section 4.5). Church published his paper on 15 April 1936, about one month before Turing submitted his paper on 28 May 1936. In his paper, Turing also proved the equivalence of his machine to Church’s \( \lambda \)-calculus. Interestingly, Turing’s paper was submitted only a few months before Emil Post, another great logician, submitted a paper independently proposing an almost identical computational model on 7 October 1936 \[62\]. The major difference between Turing’s and Post’s machines is that the former uses a tape which is infinite in only one direction, while the latter works with a tape which is infinite at both ends, like the “Turing machine” that we used in this section. We actually took our definitions in this section from Rogers’ book \[67\], which we recommend the reader for more details in addition to comprehensive textbooks such as Sipser \[76\] and Hopcroft, Motwani and Ullman \[33\]. To remember the fact that Post and Turing independently invented an almost identical computational model of utmost importance, several computer scientists call it the “Post-Turing machine”.

Even though Turing was not the first to propose what we call today a Turing-complete model of computation, many believe that his result was stronger than Church’s, in that his computational model was more direct, easier to understand, and based on first, low-level computational principles. For example, it is typically easy to implement Turing Machines in any programming language, which is not necessarily the case for other Turing-complete models, such as, for example, the \( \lambda \)-calculus. As seen in Section 4.5, \( \lambda \)-calculus relies on a non-trivial notion of substitution, which comes with the infamous variable capture problem.

### 2.2.5 Exercises

**Exercise 12.** Define Turing machines corresponding to the addition, multiplication, and power operations on natural numbers. For example, the initial configuration of the Turing machine computing addition with 3 and 7 as input is \((q_s, 0^60111101111111110^\omega)\), and its final configuration is \((q_h, 0^601111111111110^\omega)\). We here assumed that \( n \) is encoded as \( n + 1 \) cells containing 1.

**Exercise 13.** Show that there are real numbers which are not Turing computable. (Hint: The set of Turing machines is recursively enumerable.)
2.3 Propositional Logic

In this section we discuss propositional logic, often referred to as propositional calculus. Its sentences are called propositions and are constructed from propositional variables, truth values, and the usual propositional connectives. Its models are simple maps assigning truth values to propositional variables. We present a common proof system for propositional calculus and prove its soundness and completeness.

2.3.1 Syntax and Semantics

Propositional formulae, or simply propositions, are constructed using the following formal syntax:

\[
PVar \ ::= \text{propositional variables, assumed to be a countable set}
\]

\[
\phi \ ::= \ PVar \mid \bot \mid \phi \rightarrow \phi
\]

We typically use Latin letters \(p, q, r\), etc., with or without subscripts and superscripts, to refer to propositional variables in \(PVar\), and use Greek letters \(\varphi, \phi, \psi\), etc., with or without subscripts or superscripts, to refer to propositions. The formula \(\bot\) is read false, and \(\rightarrow\) is read implies. It is customary to extend the syntax of formulae with additional syntactic sugar constructs, such as the following common ones:

\[
\phi \ ::= \cdots \mid T \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \leftrightarrow \phi
\]

These can be (iteratively) desugared into the basic constructs as follows:

- (true) \(T \equiv \neg \bot\)
- (negation) \(\neg \phi \equiv \phi \rightarrow \bot\)
- (and) \(\phi_1 \land \phi_2 \equiv \neg (\neg \phi_1 \rightarrow \neg \phi_2)\)
- (or) \(\phi_1 \lor \phi_2 \equiv \neg \phi_1 \rightarrow \phi_2\)
- (equivalent) \(\phi_1 \leftrightarrow \phi_2 \equiv (\phi_1 \rightarrow \phi_2) \land (\phi_2 \rightarrow \phi_1)\)

We took \(\bot\) and \(\rightarrow\) as basic constructs here, but there are other formulations of propositional logic where other constructs are taken as basic and then desugar \(\bot\) and \(\rightarrow\) into those (e.g., \(\phi_1 \rightarrow \phi_2 \equiv \neg \phi_1 \lor \phi_2\), etc.).

A propositional model, also called a truth assignment, is a map assigning a truth value to each propositional variable. Models and propositions yield a satisfaction relation:

**Definition 9.** A propositional model is a map \(\rho : PVar \rightarrow \{\text{true, false}\}\). The satisfaction relation \(\models\) between models and propositions is defined inductively as follows:

\[
\rho \models p \iff \rho(p) = \text{true}
\]

\[
\rho \models \bot \text{ never holds, that is, } \rho \not\models \bot
\]

\[
\rho \models \phi_1 \rightarrow \phi_2 \iff \rho \not\models \phi_1 \text{ implies } \rho \models \phi_2
\]

The satisfaction relation extends to sets of propositions: if \(\Gamma\) is a set of propositions then \(\rho \models \Gamma\) iff \(\rho \models \varphi\) for each \(\varphi \in \Gamma\). If \(\Gamma\) is a set of propositions and \(\varphi\) is a proposition, then \(\Gamma\) semantically entails \(\varphi\), written \(\Gamma \models \varphi\), iff \(\rho \models \Gamma\) implies \(\rho \models \varphi\) for each model \(\rho\). A proposition \(\varphi\) is valid, or is a tautology, written \(\models \varphi\), iff \(\emptyset \models \varphi\), or equivalently, iff \(\rho \models \varphi\) for any model \(\rho\).

The following results will help us prove the soundness of our proof system for propositional logic:

**Proposition 2.** The following hold for any propositions \(\varphi_1, \varphi_2\) and \(\varphi_3\):
\[\vdash \varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_1);\]

\[\vdash (\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)) \rightarrow ((\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \varphi_3));\]

\[\vdash (\neg \varphi_1 \rightarrow \neg \varphi_2) \rightarrow (\varphi_2 \rightarrow \varphi_1);\]

If \(\vdash \varphi_1\) and \(\vdash \varphi_2\) then \(\vdash \varphi_2\).

**Proof.** Left as an exercise to the reader (Exercise 14). \(\square\)

### 2.3.2 Proof System and Soundness

Consider the following proof system based on the properties proved in Proposition 2:

\[
\begin{align*}
(Axiom_1) & \quad \varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_1) \\
(Axiom_2) & \quad (\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)) \rightarrow ((\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \varphi_3)) \\
(Axiom_3) & \quad (\neg \varphi_1 \rightarrow \neg \varphi_2) \rightarrow (\varphi_2 \rightarrow \varphi_1) \\
(Modus Ponens) & \quad \frac{\varphi_1, \varphi_1 \rightarrow \varphi_2}{\varphi_2}
\end{align*}
\]

As explained in Section 2.1.5, the above are in fact axiom and proof rule schemata, that is, each stands for infinitely many instances of the metavariables \(\varphi_1, \varphi_2\) and \(\varphi_3\).

We let \(\Gamma, \Gamma', \Gamma_0, \text{ etc.},\) range over sets of propositions. Following the general notation and terminology in Section 2.1.5, \(\Gamma \vdash \varphi\) denotes that \(\varphi\) can be derived from the set of propositions \(\Gamma\) using the proof system above. Also, \(\vdash \varphi\) is a shorthand for \(\emptyset \vdash \varphi\), and such a proposition \(\varphi\) is called a *theorem* of propositional logic.

**Theorem 1.** *(Soundness)* \(\Gamma \vdash \varphi\) implies \(\Gamma \models \varphi\). In particular, \(\vdash \varphi\) implies \(\models \varphi\) (when \(\Gamma = \emptyset\)).

**Proof.** Follows easily by structural induction on the proof derivation tree of \(\Gamma \vdash \varphi\), using Proposition 2. \(\square\)

**Proposition 3.** \(\varphi \rightarrow \varphi, \top, \varphi \rightarrow \top,\) and \(\bot \rightarrow \varphi\) are all theorems of propositional logic.

**Proof.** For \(\vdash \varphi \rightarrow \varphi\), use an *Axiom* 2 instance with \(\varphi_1 \mapsto \varphi\) and \(\varphi_2 \mapsto (\varphi \rightarrow (\varphi \rightarrow \varphi))\) and \(\varphi_3 \mapsto \varphi\), followed by two applications of *Modus Ponens* with two different instances of *Axiom* 1. Since \(\top \equiv \bot \rightarrow \bot, \vdash \top\) can be derived similarly to how \(\vdash \varphi \rightarrow \varphi\) was derived above. \(\vdash \varphi \rightarrow \top\) follows by *Modus Ponens* from \(\vdash \top\) and an *Axiom* 1 instance with \(\varphi_1 \mapsto \top\) and \(\varphi_2 \mapsto \varphi\). Finally, \(\vdash \bot \rightarrow \varphi\) follows by *Modus Ponens* from \(\vdash \neg \varphi \rightarrow \bot\) (which can be derived similarly to \(\vdash \varphi \rightarrow \top\)) and an instance of *Axiom* 3 with \(\varphi_1 \mapsto \varphi\) and \(\varphi_2 \mapsto \bot\). \(\square\)

**Theorem 2.** *(Deduction)* \(\Gamma \vdash \varphi_1 \rightarrow \varphi_2\) iff \(\Gamma \cup \{\varphi_1\} \vdash \varphi_2\).

**Proof.** If \(\Gamma \vdash \varphi_1 \rightarrow \varphi_2\) then \(\Gamma \cup \{\varphi_1\} \vdash \varphi_2\) follows easily by *Modus Ponens*. For the other implication, we show by structural induction on the proof derivation tree that for any derived formula \(\varphi\) in the derivation tree of \(\Gamma \cup \{\varphi_1\} \vdash \varphi_2\) we can also derive \(\Gamma \vdash \varphi_1 \rightarrow \varphi_2\). The result will then follow by taking \(\varphi\) to be \(\varphi_2\).

If \(\varphi\) is an axiom (instance) or a formula in \(\Gamma\), then we can use *Axiom* 1 instantiated with \(\varphi_1 \mapsto \varphi\) and \(\varphi_2 \mapsto \varphi_1\), followed by *Modus Ponens* instantiated with \(\varphi_1 \mapsto \varphi\) and \(\varphi_2 \mapsto (\varphi_1 \rightarrow \varphi)\), and derive \(\Gamma \vdash \varphi_1 \rightarrow \varphi\).

If \(\varphi\) is \(\varphi_1\) then \(\Gamma \vdash \varphi_1 \rightarrow \varphi\) by Proposition 3.

If \(\varphi\) is derived using *Modus Ponens* from previously derived formulae \(\psi\) and \(\psi \rightarrow \varphi\), then \(\Gamma \vdash \varphi_1 \rightarrow \psi\) and \(\Gamma \vdash \varphi_1 \rightarrow (\psi \rightarrow \varphi)\) are derivable (the induction hypothesis), so we can also derive \(\Gamma \vdash \varphi_1 \rightarrow \varphi\) by *Axiom* 2 with instance \(\varphi_1 \mapsto \varphi_1\) and \(\varphi_2 \mapsto \psi\) and \(\varphi_3 \mapsto \varphi\), using *Modus Ponens* twice. \(\square\)
Proposition 4. \( \vdash \varphi \rightarrow \neg \neg \varphi \) and \( \vdash \neg \neg \varphi \rightarrow \varphi \).

Proof. \( \{ \varphi, \varphi \rightarrow \bot \} \vdash \bot \) by Modus Ponens. Then the Deduction Theorem applied twice yields \( \vdash \varphi \rightarrow \neg \neg \varphi \). We can similarly show \( \vdash \neg \varphi \rightarrow \neg \neg \neg \varphi \), which together with Axiom \( \text{Axiom}_3 \) instantiated with \( \varphi_1 \leftrightarrow \varphi \) and \( \varphi_2 \leftrightarrow \neg \neg \neg \varphi \), yields \( \vdash \neg \neg \varphi \rightarrow \varphi \) by Modus Ponens. \( \square \)

**Corollary 1.** \( \Gamma \vdash \varphi \) iff \( \Gamma \vdash \neg \neg \varphi \) iff \( \Gamma \cup \{ \neg \varphi \} \vdash \bot \).

Proof. Deduction Theorem implies \( \Gamma \cup \{ \neg \varphi \} \vdash \bot \) iff \( \Gamma \vdash \neg \varphi \rightarrow \bot \). The rest follows by Proposition 4. \( \square \)

### 2.3.3 Consistency and Model Existence

Consistency is a crucial instrument for the proof of the Completeness Theorem in Section 2.3.4. A set of propositions is consistent iff it cannot be used to derive \( \bot \). The main result of this section states that a set of propositions is consistent iff it admits a model.

**Definition 10.** \( \Gamma \) is **inconsistent** iff \( \Gamma \vdash \bot \), and is **consistent** iff \( \Gamma \nvdash \bot \). Moreover, \( \Gamma \) is **maximally consistent** iff (1) \( \Gamma \) is consistent and (2) \( \Gamma \subseteq \Gamma' \) and \( \Gamma' \) consistent imply \( \Gamma = \Gamma' \).

**Proposition 5.** \( \Gamma \) is inconsistent iff \( \Gamma \vdash \varphi \) for any proposition \( \varphi \).

Proof. If \( \Gamma \vdash \varphi \) for any proposition \( \varphi \), then also \( \Gamma \vdash \bot \), so \( \Gamma \) inconsistent. If \( \Gamma \vdash \bot \) then since \( \bot \rightarrow \varphi \) for any \( \varphi \) (by Proposition 3), it follows by Modus Ponens that \( \Gamma \vdash \varphi \) for any proposition \( \varphi \). \( \square \)

**Proposition 6.** Suppose that \( \Gamma \) is consistent and \( \varphi \) is any proposition. Then:

1. \( \Gamma \cup \{ \varphi \} \) is consistent, or \( \Gamma \cup \{ \neg \varphi \} \) is consistent, or both;

2. If \( \Gamma \) is maximally consistent, then either \( \varphi \in \Gamma \) or \( \neg \varphi \in \Gamma \).

Proof. For the first part, if both \( \Gamma \cup \{ \varphi \} \) and \( \Gamma \cup \{ \neg \varphi \} \) are inconsistent, then \( \Gamma \vdash \neg \varphi \) and \( \Gamma \vdash \neg \varphi \rightarrow \bot \) by Deduction Theorem so Modus Ponens implies \( \Gamma \) inconsistent. The second is immediate from the first. \( \square \)

**Corollary 2.** Suppose that \( \Gamma \) is maximally consistent and \( \varphi \) is any proposition. Then \( \Gamma \vdash \varphi \) iff \( \varphi \in \Gamma \).

Proof. If \( \varphi \in \Gamma \) then obviously \( \Gamma \vdash \varphi \). Conversely, suppose \( \Gamma \vdash \varphi \). If \( \varphi \notin \Gamma \) then \( \neg \varphi \in \Gamma \) by Proposition 6 which together with \( \Gamma \vdash \varphi \) yield \( \Gamma \vdash \bot \) by Modus Ponens, which contradicts the consistency of \( \Gamma \). \( \square \)

**Proposition 7.** If \( \Gamma \) is maximally consistent, then \( \varphi_1 \rightarrow \varphi_2 \in \Gamma \) iff \( \varphi_1 \in \Gamma \) implies \( \varphi_2 \in \Gamma \).

Proof. \( \varphi_1 \rightarrow \varphi_2 \in \Gamma \) and \( \varphi_1 \in \Gamma \), then \( \varphi_2 \in \Gamma \) by Modus Ponens and Corollary 2. Conversely, suppose that \( \varphi_1 \in \Gamma \) implies \( \varphi_2 \in \Gamma \). We can distinguish two cases: (1) \( \varphi_2 \in \Gamma \); and (2) \( \varphi_1 \notin \Gamma \), that is, \( \neg \varphi_1 \in \Gamma \) (by Proposition 6). In the first case, \( \varphi_1 \rightarrow \varphi_2 \in \Gamma \) by Modus Ponens with an instance of Axiom 1 with \( \varphi_1 \leftrightarrow \varphi_2 \) and \( \varphi_2 \leftrightarrow \varphi_1 \), and by Corollary 2. In the second case, \( \varphi_1 \rightarrow \varphi_2 \in \Gamma \) by Modus Ponens with the theorem \( \neg \varphi_1 \rightarrow (\varphi_1 \rightarrow \varphi_2) \) and by Corollary 2 again. That \( \neg \varphi_1 \rightarrow (\varphi_1 \rightarrow \varphi_2) \) is a theorem follows easily by Deduction Theorem and \( \vdash \bot \rightarrow \varphi_2 \) (which follows by Proposition 5). \( \square \)

**Theorem 3.** *(Lindenbaum)* \( \Gamma \) consistent implies there exists some \( \Gamma' \) maximally consistent such that \( \Gamma \subseteq \Gamma' \).
Proof. Since the set of propositional variables $PVar$ is countable, the entire (infinite) set of propositions is countable. Let us enumerate them in some arbitrary but fixed order, say $\varphi_1, \varphi_2, \ldots$, and let us build the (infinite) sequence of consistent sets of propositions $\Gamma_0 \subseteq \Gamma_1 \subseteq \Gamma_2 \subseteq \ldots$ as follows: $\Gamma_0 = \Gamma$ and for each $n > 0$, $\Gamma_n$ is one of $\Gamma_{n-1} \cup \{\varphi\}$ or $\Gamma_{n-1} \cup \{\neg \varphi\}$ which is consistent; if both are consistent, then we arbitrarily pick one. Since $\Gamma_0$ is consistent, by Proposition 6 it inductively follows that each of $\Gamma_n$ is consistent.

Let $\Gamma'$ be $\bigcup_{n \geq 0} \Gamma_n$. We claim that $\Gamma'$ is consistent. Indeed, if $\Gamma' \vdash \bot$ then there is a finite proof derivation of $\bot$ from $\Gamma'$, involving only a finite number of propositions in $\Gamma'$ (by Proposition 1). In other words, there is some $n \geq 0$ such that all those propositions are in $\Gamma_n$, which implies $\Gamma_n \vdash \bot$, which contradicts $\Gamma_n$ consistent. We claim that $\Gamma'$ is also maximal. Indeed, suppose that there is some $\varphi$ such that $\varphi \notin \Gamma'$ and $\Gamma' \cup \{\varphi\}$ is consistent. Since we enumerated all propositions, there exists an $n \geq 0$ such that $\varphi = \varphi_n$. Since $\varphi_n \notin \Gamma'$, it follows that $\varphi_n \notin \Gamma_n$, so it must be that $\neg \varphi_n \in \Gamma_n$, that is, $\neg \varphi \in \Gamma'$ (since $\varphi = \varphi_n$ and $\Gamma_n \subseteq \Gamma'$), which contradicts $\Gamma' \cup \{\varphi\}$ consistent.

**Theorem 4. (Model Existence)** $\Gamma$ consistent iff $\rho \models \Gamma$ for some model $\rho$.

Proof. If $\rho \models \Gamma$ for some model $\rho$, then $\Gamma$ must be consistent: otherwise, if $\Gamma \vdash \bot$, then by Soundness Theorem 1 we get $\Gamma \vdash \bot$, which leads to a contradiction, namely $\rho \models \bot$.

Conversely, suppose that $\Gamma$ is consistent. Let $\Gamma'$ be a maximally consistent set with $\Gamma \subseteq \Gamma'$, which exists thanks to Lindenbaum Theorem 3. Let us define a model $\rho : PVar \to \{true, false\}$ as follows: for each $p \in PVar$, $\rho(p) = true$ iff $p \in \Gamma'$. Proposition 6 implies that $\rho$ is indeed well-defined.

To show $\rho \models \Gamma$, we show by structural induction on $\varphi$ the stronger result “$\rho \models \varphi$ iff $\varphi \in \Gamma'$”. The cases $\varphi \in PVar$ and $\varphi \equiv \bot$ are trivial. If $\varphi \equiv \varphi_1 \rightarrow \varphi_2$ then $\varphi \in \Gamma'$ iff (by Proposition 7) $\varphi_1 \in \Gamma'$ implies $\varphi_2 \in \Gamma'$, iff (by the induction hypothesis) $\rho \models \varphi_1$ implies $\rho \models \varphi_2$, iff $\rho \models \varphi$.

### 2.3.4 Completeness and Compactness

The Completeness Theorem below states that the three axiom schemata together with *Modus Ponens* are in fact capable of deriving any tautology of propositional calculus. We prefer to present the Compactness Theorem as an immediate corollary of the Soundness Theorem 1 and the Completeness Theorem 5, although the reader should note that in other places it is presented as a corollary of the Model Existence Theorem 4.

**Theorem 5. (Completeness)** $\Gamma \models \varphi$ implies $\Gamma \vdash \varphi$. In particular, $\models \varphi$ implies $\vdash \varphi$ (when $\Gamma = \emptyset$).

Proof. Suppose that $\Gamma \models \varphi$ and $\Gamma \not\models \varphi$. Then $\Gamma \cup \{\neg \varphi\}$ is consistent due to Corollary 1, so Theorem 1 implies that $\rho \models \Gamma$ and $\rho \not\models \varphi$ for some model $\rho$, which contradicts $\Gamma \models \varphi$.

**Theorem 6. (Compactness)** If $\Gamma \models \varphi$ then there is a finite subset $\Gamma_0$ of $\Gamma$ such that $\Gamma_0 \models \varphi$.

Proof. If $\Gamma \models \varphi$ then $\Gamma \vdash \varphi$ by the Completeness Theorem (Theorem 5). By Proposition 1 there is some finite subset $\Gamma_0$ of $\Gamma$ such as $\Gamma_0 \vdash \varphi$. Finally, by the Soundness Theorem (Theorem 1) it follows that $\Gamma_0 \models \varphi$.

### 2.3.5 Notes

Propositional logic evolved over a period of more than 2000 years from general advances in philosophy, mathematics, and logic, where a basic calculus with propositions and truth values could simply not be avoided. For example, the Greek Stoics were well-known for their use of rigorous logic in the 3rd century BC. Although the origins of propositional logic as we know it today can be and have been pushed back to the 17th century to Gottfried Wilhelm Leibniz, and even to the 12th century to Peter Abelard, it is generally accepted now.
that the fathers of propositional logic are George Boole (1815-1864) and Augustus De Morgan (1806-1871). Jan Łukasiewicz (1878-1956), nevertheless, has made many important contributions to propositional logic, including the proof system that we presented in this section. There are many other proof systems for propositional logics in the literature, several proposed by Łukasiewicz himself, but their soundness and completeness can be shown following a similar pattern. The general proof style based on a model existence theorem that we followed in this section, which also applies to more complex logics, is due to Leon Albert Henkin (1921-2006). The Lindenbaum theorem, which is critical for the proof of the model existence theorem, was attributed to Adolf Lindenbaum (1904-1941) by Alfred Tarski (1901-1983).

2.3.6 Exercises

The following exercises complete the material on propositional logic in Sections 2.3.1, 2.3.2, and 2.3.3.


Exercise 15. Using only the basic notions in Section 2.3.1 prove that $\Gamma \models \varphi_1 \rightarrow \varphi_2$ iff $\Gamma \cup \{\varphi_1\} \models \varphi_2$.

Exercise 16. Using only the basic notions in Section 2.3.1 prove that the theorems listed in Proposition 3 are all tautologies.

Exercise 17. Give complete proof derivations for the propositional logic theorems in Proposition 3.

Exercise 18. Prove $\vdash \neg \varphi_1 \rightarrow (\varphi_1 \rightarrow \varphi_2)$, both directly using the proof system and indirectly using the Deduction Theorem (Theorem 2). This theorem is used in Proposition 7 which can be consulted for a hint.

Exercise 19. Prove $\vdash (\varphi_1 \rightarrow \varphi_2) \rightarrow ((\varphi_2 \rightarrow \varphi_3) \rightarrow (\varphi_1 \rightarrow \varphi_3))$ and $\vdash (\varphi_1 \rightarrow \varphi_2) \rightarrow (\neg \varphi_2 \rightarrow \neg \varphi_1)$.

Exercise 20. If $\Gamma$ is maximally consistent then:

- $\varphi_1 \land \varphi_2 \in \Gamma$ iff $\varphi_1 \in \Gamma$ and $\varphi_2 \in \Gamma$; and
- $\varphi_1 \lor \varphi_2 \in \Gamma$ iff $\varphi_1 \in \Gamma$ or $\varphi_2 \in \Gamma$.
- $\varphi_1 \leftrightarrow \varphi_2 \in \Gamma$ iff $\varphi_1 \in \Gamma$ iff $\varphi_2 \in \Gamma$.

Should we move after the RL section and add one or two ACI rewriting decision procedures for propositional logic? Similar question for FOL, once added.
2.4 Equational Logic

define lists, sets, bags

say that we allow Boolean terms \( b \) in conditions of equations and rules, as syntactic sugar for \( b = \text{true} \)

add the \( \Pi_2^0 \) result for initial model satisfaction here: grab material from the corresponding directory under “Current Work”.

add a subsection on \( \Sigma \)-algebras

say this once and for all in this section when discussing the property of the initial algebra, and make sure that it does not repeat in the rest of the book: “we let \( \theta \) also denote the homomorphic extension \( T_\Sigma(\mathcal{X}) \rightarrow T_\Sigma \) of \( \theta : \mathcal{X} \rightarrow T_\Sigma \) to terms with variables in \( \mathcal{X} \)”

use calligraphic font as above for sets, as we use \( \mathcal{X} \) for one variable in Maude specs.

algebraic specification, equational theory, equational specification

2.4.1 Signatures, Algebras, Equations, Satisfaction

...

2.4.2 Proof System and Soundness

...
sorts:
   Cell, Tape, Configuration

operations:
   0, 1 : → Cell
   zeros : → Tape
   _ : _ : Cell × Tape → Tape
   q : Tape × Tape → Configuration — one such operation for each q ∈ Q

generic equation:
   zeros = 0 : zeros

specific equations:
   q(L, b :R) = q'(L, b':R) — one equation for each q, q' ∈ Q, b, b' ∈ Cell with M(q, b) = (q', b')
   q(L, b :R) = q'(b :L, R) — one equation for each q, q' ∈ Q, b ∈ Cell with M(q, b) = (q', →)
   q(B :L, b :R) = q'(L, B : b :R) — one equation for each q, q' ∈ Q, b ∈ Cell with M(q, b) = (q', ←)

Figure 2.2: Lazy equational logic representation $E_{lazy}^M$ of Turing machine $M$

2.4.3 Computation as Equational Deduction

Here we discuss simple equational logic encodings of Turing machines (see Section 2.2.1 for general Turing machine notions). The idea is to associate an equational theory to any Turing machine, so that an input is accepted by the Turing machine if and only if an equation corresponding to that input can be proved from the equational theory of the Turing machine, using conventional equational deduction. Moreover, as seen in Section 2.5.3, the resulting equational theories can be executed as rewrite theories by rewrite engines, thus yielding actual Turing machine interpreters. We present two encodings, both based on intuitions from lazy data-structures, specifically stream data-structures. The first is simpler but requires lazy rewriting support from rewrite engines in order to be executed, while the second can be executed by any rewrite engines.

Lazy Equational Representation

Our first representation of Turing machines in equational logic is based on the idea that the infinite tape can be finitely represented by means of self-expanding stream data-structures. In spite of being infinite sequences of cells, like the Turing machine tapes, many interesting streams can be finitely specified using equations. For example, the stream of zeros, $zeros = 0 : 0 : 0 : \cdots$, can be defined as $zeros = 0 : zeros$. Since at any given moment the portions of a Turing machine tape to the left and to the right of the head have a suffix consisting of an infinite sequence of 0 cells, it is natural to represent them as streams of the form $b_1 : b_2 : \cdots : b_n : zeros$. When the head is on cell $b_n$ and the command is to move the head to the right, the self-expanding equational definition of $zeros$ can produce one more 0, so that the head can move onto it. To expand $zeros$ on a by-need basis and thus to avoid undesired non-termination due to the uncontrolled application of the self-expanding equation of $zeros$, this approach requires an equational/rewrite engine with support for lazy evaluation/rewriting in order to be executed.

Figure 2.2 shows how a Turing machine $M = (Q, B, q_s, q_h, C, M)$ can be associated a computationally equivalent equational logic theory $E_{lazy}^M$. Except for the self-expanding equation of the $zeros$ stream and our stream representation of the two-infinite-end tape, the equations of $E_{lazy}^M$ are identical to the transition relation on Turing machine configurations discussed right after Definition 7. The self-expanding equation of $zeros$
Theorem 7. The following are equivalent:

(1) The Turing machine $M$ terminates on input $b_1b_2 \ldots b_n$;

(2) $E^\text{lacy}_M \models q_s(\text{zeros}, b_1 : b_2 : \cdots : b_n : \text{zeros}) = q_h(l, r)$ for some terms $l, r$ of sort Tape.

Proof. \ldots \hfill \square

The equations in $E^\text{lacy}_M$ can be applied in any direction, so an equational proof of $E^\text{lacy}_M \models q_s(\text{zeros}, b_1 : b_2 : \cdots : b_n : \text{zeros}) = q_h(l, r)$ needs not necessarily correspond step-for-step to the computation of $M$ on input $b_1b_2 \ldots b_n$. We will see in Section 2.5.3 that by orienting the specific equations in Figure 2.2 into rewrite rules, we will obtain a rewrite logic theory which will faithfully capture, step-for-step, the computational granularity of $M$.

Note that in Figure 2.2 we preferred to define a configuration construct $q : \text{Tape} \times \text{Tape} \to \text{Configuration}$ for each $q \in Q$. A natural alternative could have been to define an additional sort State for the Turing machine states, a constant $q : \to \text{State}$ for each $q \in Q$, and one generic configuration construct $\_ : \text{State} \times \text{Tape} \times \text{Tape} \to \text{Configuration}$, as we do in the subsequent representation of Turing machines as rewrite logic theories (see Figure 2.3). The reason for which we did not do that here is twofold: first, in functional languages like Haskell it is very natural to associate a function to each such configuration construct which can be applied in any direction, so an equational proof of $E^\text{lacy}_M \models \text{zeros} = \text{zeros}$ from the representation in Figure 2.2 and replace it with the two safe equations in Figure 2.3. Let $E_M$ be the equational logic theory in Figure 2.3.

Theorem 8. The following are equivalent:
sorts:
  Cell, Tape, State, Configuration

operations:
0, 1 : → Cell
zeros : → Tape
- : : Cell × Tape → Tape
\langle-,-\rangle : State × Tape × Tape → Configuration
q : → State — one such constant for each q ∈ Q

generic equations:
S(zeros, R) = S(0:zeros, R)
S(L, zeros) = S(L, 0:zeros)

specific equations:
q(L, b : R) = q'(L, b':R) — one equation for each q, q' ∈ Q, b, b' ∈ Cell with M(q, b) = (q', b')
q(L, b : R) = q'(b : L, R) — one equation for each q, q' ∈ Q, b ∈ Cell with M(q, b) = (q', →)
q(B : L, b : R) = q'(L, B : b : R) — one equations for each q, q' ∈ Q, b ∈ Cell with M(q, b) = (q', ←)

Figure 2.3: Unrestricted equational logic representation E_M of Turing machine M

(1) The Turing machine M terminates on input b_1 b_2 ... b_n;
(2) E_M |= q_{s}(zeros, b_1 : b_2 : ... : b_n; zeros) = q_{h}(l, r) for some terms l, r of sort Tape.

Proof. ... □

One could argue that deduction with the equational theory in Figure 2.3 is not fully faithful to computations with the original Turing machine, because the two generic equations may need to artificially apply from time to time as an artifact of our representation, and their application does not correspond to actual computational steps in the Turing machine. In fact, these generic equations can be completely eliminated, at the expense of more equations. For example, if M(q, b) = (q', ←) then, in addition to the last equation in Figure 2.3 we can also include the equation:

q(zeros, b : R) = q'(zeros, 0 : b : R)

This way, one can expand zeros and apply the transition in one equational step. Doing that systematically for all the transitions allows us to eliminate the need for the two generic equations entirely.
2.4.4 Initial Algebra

Why are they called “initial” models?

The next result shows that the problem of equational satisfaction in an initial algebra is a \( \Pi^0_2 \)-complete problem. That tells us that there is no way to automate the process of proving equalities or inequalities in initial algebras in general; in particular, induction cannot be automated in general.

**Theorem 9.** Equational satisfaction in an initial algebra is a \( \Pi^0_2 \)-complete problem.

**Proof.** Let us first show the membership in \( \Pi^0_2 \) part. We have to show that the problem taking as input an equational specification \((\Sigma, E)\) and a (not necessarily ground) \(\Sigma\)-equation \(e\) and saying whether \(T_{\Sigma,E} \models e\) is of the form \(\forall i(\exists j) \, r(i, j, k)\), where \(i\) and \(j\) range over natural numbers, \(k\) is the Gödel number corresponding to the input \((\Sigma, E)\) and \(e\), and \(r\) is some recursive, or decidable, property. Let us fix an input to the initial algebra problem, that is, an equational specification \((\Sigma, E)\) and a \(\Sigma\)-equation \(e\), say \((\forall X) \, t = t'\).

Since we can enumerate, or Gödelize, all mappings \(X \to T_{\Sigma,E}\), there is a one-to-one correspondence between natural number indexes \(i\) and mappings \(\theta : X \to T_{\Sigma,E}\). Let \(i_\theta\) denote the index corresponding to substitution \(\theta\) and let \(\theta_i\) denote the \(i\)-th substitution; then obviously \(i_{\theta_i} = i\) and \(\theta_{i_\theta} = \theta\) for any \(i\) and \(\theta\). The \(\Sigma\)-terms \(t\) and \(t'\) over variables in \(X\) are taken by any map \(\theta : X \to T_{\Sigma,E}\) into equivalence classes of ground terms in \(T_{\Sigma,E}\). Therefore, \(T_{\Sigma,E} \models e\) iff \(\theta(t) \equiv_E \theta(t')\) for any mapping \(\theta : X \to T_{\Sigma,E}\). However, recall that \(\theta(t) \equiv_E \theta(t')\) iff \(E \vdash (\forall \theta) \theta(t) = \theta(t')\), that is, iff and only if there exists some proof derivation of \((\forall \theta) \theta(t) = \theta(t')\) from \(E\) using the equational proof system. Since we can enumerate all the proofs derived from an r.e. set of equations \(E\), there is a one-to-one correspondence between natural number indexes \(j\) and proof derivations. Let \(j_\pi\) denote the index corresponding to proof \(\pi\) and let \(\pi_j\) denote the \(j\)-th proof; then obviously \(j_{\pi_j} = j\) and \(\pi_{j_\pi} = \pi\) for any \(j\) and \(\pi\). Therefore,

\[
T_{\Sigma,E} \models e \iff (\forall i)(\exists j) \, r(i, j, k),
\]

where \(i\) ranges over mappings \(\theta : X \to T_{\Sigma,E}\), \(j\) over equational proofs, and \(r(i, j, k)\) is the test whether proof \(\pi_j\) is correct for the task \(E \vdash (\forall \theta) \theta(t) = \theta(t')\); since proof checking is decidable, it follows that the equational satisfaction problem in an initial model is a \(\Pi^0_2\) problem.

Let us now focus on the \(\Pi^0_2\)-hardness part. It suffices to build a particular finite equational specification \(\mathcal{E} = (\Sigma, E)\) and a \(\Sigma\)-equation \(e\), such that \(T_{\Sigma,E} \models e\) is a \(\Pi^0_2\)-hard problem; note that \(e\) must have variables (the satisfaction problem for ground equations is no different in initial algebras than in general, so it is r.e. – see Proposition 8 and the discussion following it). Let us start with \(\mathcal{E}\); we build it as an extension of \(\mathcal{E}_U\), the equational specification corresponding to a Turing machine constructed following the general procedure in Section ?? but applied to \(\mathcal{U}\), the universal Turing machine in Section ?? that makes the \(\textsc{Totality}\) problem \(\Pi^0_3\)-complete. Let \(\mathcal{E} = (\Sigma, E)\) be the finite equational specification obtained by adding to \(\mathcal{E}_U\) the following:

- A sort \(\text{Nat}\), a constant \(\text{zero} :\to \text{Nat}\) and a unary operation \(\text{succ} : \text{Nat} \to \text{Nat}\);

- An operation \(\text{ones} : \text{Nat} \times \text{List} \to \text{List}\) together with equations

\[
(\forall L : \text{List}) \, \text{ones}(\text{zero}, L) = L,
\]

\[
(\forall N : \text{Nat}, L : \text{List}) \, \text{ones}(\text{succ}(N), L) = \text{ones}(N, 1 : L);
\]

- An operation \(u : \text{Nat} \times \text{Nat} \to \text{State}\) together with the equation

\[
(\forall J, K : \text{Nat}) \, u(J, K) = q_s(\text{ones}(J, 0 : \text{ones}(K, \text{nil}))).
\]
Then by Proposition ?? and 1 in Proposition[8] the Totality problem is equivalent to the following problem, where \( \overline{n} \) is the Peano representation \( \text{succ}(\cdots(\text{succ}(\text{zero})\cdots) \) of the natural number \( n \), that is, the term of sort \( \text{Nat} \) consisting of a sequence of \( n \) successor operators applied on \( \text{zero} \):

\[
\text{INPUT: An integer } k \geq 0; \\
\text{OUTPUT: Does } E \models (\forall \theta)\ u(\overline{j}, \overline{k}) = \bot \text{ for all } j \geq 0?
\]

We would like to transform the infinitary conjunction of satisfaction relations in the output of the problem above into only one satisfaction relation, but in an initial model. We claim that the problem above is entirely equivalent to the following problem:

\[
\text{INPUT: An integer } k \geq 0; \\
\text{OUTPUT: Does } T_{\Sigma, E} \models (\forall J : \text{Nat})\ u(J, \overline{k}) = \bot?
\]

Indeed,

\[
\begin{align*}
T_{\Sigma, E} \models (\forall J : \text{Nat})\ u(J, \overline{k}) = \bot & \iff \\
T_{\Sigma, E} \models (\forall \theta)\ \theta(u(\overline{j}, \overline{k})) = \bot \text{ for all } \theta : J \rightarrow T_{\Sigma, E} & \iff \\
T_{\Sigma, E} \models (\forall \theta)\ (u(\overline{j}, \overline{k})) = \bot \text{ for all } j \geq 0 & \iff \\
E \models (\forall \theta)\ u(\overline{j}, \overline{k}) = \bot \text{ for all } j \geq 0 & \text{(by 1 in Proposition[8]).}
\end{align*}
\]

Therefore, the satisfaction problem in an initial algebra is at least as hard as the Totality problem, \( \Pi^0_2 \). Since we proved that it is in \( \Pi^0_2 \), it follows that it is a \( \Pi^0_2 \)-complete problem. \( \square \)

### 2.4.5 Completeness

... Let \( \Sigma \) be a many-sorted signature which contains recursively enumerable (r.e.) sets of sorts and of operations, the latter of finite arity. Let \( T_\Sigma \) denote the initial \( \Sigma \)-algebra of ground (i.e., containing no variables) \( \Sigma \)-terms; if \( \Sigma \) contains no constants then \( T_\Sigma \) is obviously empty. Let \( \text{Var} \) be a set of variables that is infinite and r.e. on each sort that appears in \( \Sigma \). If \( X \subseteq \text{Var} \) is a set of variables then we let \( \Sigma \cup X \) denote the signature \( \Sigma \) extended with the variables in \( X \) regarded as constants. Also, let \( T_\Sigma(X) \) denote the free algebra of \( \Sigma \)-terms over variables in \( X \), that is, \( T_{\Sigma \cup X} \). Equations are triples \( (\forall X)\ t = t' \), where \( t \) and \( t' \) are \( \Sigma \)-terms over variables in \( X \). If \( X \) is empty then the equation is called ground. Let \( (\Sigma, E) \) be any equational theory where \( E \) is an r.e. set of \( \Sigma \)-equations. We assume that all equations are unconditional but can contain variables. We let \( T_{\Sigma, E} \) denote the initial \((\Sigma, E)\)-algebra. Recall that the initial \((\Sigma, E)\)-algebra contains equivalence classes of ground terms as elements, where two terms are equivalent iff they can be proved, or derived equal using \( E \); in other words, \( T_{\Sigma, E} \) is the quotient of \( T_\Sigma \) by \( \equiv_E \), where \( t \equiv_E t' \) iff \( E \vdash (\forall \theta)\ t = t' \), where \( \vdash \) is the standard equational derivability relation.

**Proposition 8. [27]** If \( e \) is any \( \Sigma \)-equation then the following hold:

1. (Soundness and completeness) \( E \models e \iff T_{\Sigma \cup X, E} \models e \iff E \vdash e \);

2. (Initial algebra semantics) If \( e \) is ground then \( E \models e \iff T_{\Sigma, E} \models e \).

The first item above tells us that the equational satisfaction problem is r.e.. Indeed, one can use the deduction system of equational logics to enumerate all proofs; the completeness theorem of equational logics ensures that this enumeration process eventually reaches any semantic equational consequence of the equational theory, with or without variables. The second item, which is a corollary of the first when \( X \) is
empty, tells us that the initial algebra of an equational theory “knows everything” about ground equalities; this is not surprising, because the initial algebra is built by collapsing ground terms that can be shown equal. An immediate consequence of the two items above is that the problem of ground satisfaction in an initial algebra is r.e.

**Pitfall.** One should not get tricked by the results above and conclude that the initial algebra can simply replace the original equational theory, thinking that any property can be tested for satisfaction now in the initial algebra instead of the original theory. That only holds for ground equalities! The initial algebra, of course, satisfies all the equational consequences of the original theory, but it can and typically does also satisfy many other non-ground equational properties. For example, the initial algebra of the Peano equational theory of natural numbers, i.e., the actual natural numbers, satisfies the commutativity of addition; nevertheless, there are many other algebras of that equational theory that do not satisfy commutativity of addition. There is indeed an initial algebra that captures precisely all the equational properties of $E$ quantified over variables in $X$, but that initial algebra is one over a larger signature, namely $T_{\Sigma \cup X \cup E}$, not $T_{\Sigma \cup E}$. Unfortunately, adding constants to an initial algebra significantly reduces the set of properties that hold in that algebra; in particular, the commutativity of addition does not hold in the algebra of natural numbers extended with two constants.
On Models

From now on we may use the word model to refer to implementations. This terminology comes from mathematical logics and model theory, so it has a more mathematical flavor. Anyhow, we should think of both implementations and models as realizations of specifications.

A specification can have therefore several models: all those satisfying its properties. However, not all models are always intended. In this lecture we discuss initial models, for which induction is a valid proof technique, and then we discuss the relationship between these and recursively defined operations.

Bad Models

Among the models of a specification, there are some which have quite unexpected properties. For example, a simple specification of lists of bits without any equations, can be defined as follows:

```plaintext
mod BIT-LIST is
  sorts Bit BitList . subsort Bit < BitList .
  ops 0 1 : -> Bit .
  op nil : -> BitList .
  op _,_ : Bit BitList -> BitList .
endm
```

One possible model, \( M \), can do the following:

- Implement elements of sort BitList as real numbers (yes, it looks strange but there is nothing to forbid this so far),
- Implement nil and 0 as 0, and 1 as 1,
- Implement \( _,_ \) as addition.

The model \( M \) has very strange properties, such as

Junk

There are lots of lists in \( M \) which are not intended to be lists of bits, such as, for example, the number \( \pi \).

Confusion

There are distinct lists which in fact collapse when interpreted in \( M \), such as, for example, the lists \( 0,1 \) and \( 1,0 \). Concatenation becomes commutative in \( M \), which is highly undesirable.

Initial Models

Initial models are those with no junk and no confusion. How can we define such a model?

For the specification of lists of bits above, we can build an initial model \( \mathcal{T} = (\mathcal{T}_{\text{Bit}}, \mathcal{T}_{\text{BitList}}) \) as the pair of smallest sets with the following properties:

1. \( 0 \) and 1 belong to \( \mathcal{T}_{\text{Bit}} \);
2. \( \mathcal{T}_{\text{Bit}} \) is included in \( \mathcal{T}_{\text{BitList}} \);
3. nil is an element of \( \mathcal{T}_{\text{BitList}} \), and \( 0, L \) and \( 1, L \) are elements of \( \mathcal{T}_{\text{BitList}} \) whenever \( L \) is an element of \( \mathcal{T}_{\text{BitList}} \).

The model \( \mathcal{T} \) is exactly the desired model for lists of bits: contains nothing but lists and no two distinct lists are collapsed!
One can similarly define an initial model for any signature. Intuitively, initial models of signatures consist of exactly all the well-formed terms over the syntax specified by the signature.

In the case of specifications, which contain not only signatures but also sentences, initial models can be defined as well. Essentially, they are defined by

taking the initial models of the corresponding signatures and collapsing all terms which can be shown equal using specification’s sentences.

As an example, let us reconsider the specification of natural numbers PEANO-NAT:

```plaintext
mod PEANO-NAT is
  sort Nat .
  op zero : -> Nat .
  op succ : Nat -> Nat .
  op plus : Nat Nat -> Nat .
  vars N M : Nat .
  eq plus(zero, M) = M .
  eq plus(succ(N), M) = succ(plus(N, M)) .
endm
```

The initial model of its signature contains all the well-formed terms built over zero, succ and plus (so no variables!). When we also consider the two equations, many distinct such terms will collapse, because they become equal modulo those equations.

E.g., plus(succ(succ(zero)), succ(succ(succ(zero)))) is equal to plus(succ(succ(succ(succ(zero)))), succ(zero)), so they represent only one element in the initial model of PEANO-NAT.

The set of all terms which are equivalent modulo the equations of a specification are typically called equivalence classes. Therefore, initial models of specifications have equivalence classes as elements.

From now on in this class, when we talk about models or implementations of specifications, we will actually mean initial models. Also, when we prove properties of specifications, we prove them as if for their initial models.

**Induction**

Why are we interested in initial models? Because in these models, induction, a very powerful proof technique very related to recursive definitions that will be intensively used in defining programming language features later in the course, is a valid proof technique.

**Where is the Mistake?**

To understand the important concept of initial model as well as its relationship to induction better, let us play a “where is the mistake” game. We prove by induction a property of a specification, and then we show that there are implementations which satisfy the specification but which do not satisfy the “proved” property.

We first prove that commutativity of \(+_\) is a consequence of the specification of Peano natural numbers, this time using the mix-fix notation:

```plaintext
mod PEANO-NAT is sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
```
vars N M : Nat .
eq \theta + N = N .
eq s(M) + N = s(M + N) .
endm

To show that $M + N = N + M$ for all natural numbers $M$ and $N$, we can do a proof by induction on either $M$ or $N$, say $M$. Since any natural number is either $\theta$ or the successor of a smaller natural number, we need to analyze two cases:

Case 1: $M = \theta$. We need to prove that $\theta + N = N + \theta$ for any natural number $N$. By the first equation, we only need to show that for any natural number $N$, it is the case that $N + \theta = N$. The only way to show it is by induction again, this time by $N$. There are two cases again:

Case 1.1: $N = \theta$. We have to show that $\theta + \theta = \theta$, which follows by the first equation.

Case 1.2: Assume $n + \theta = n$ for some $n$ and prove that $s(n) + \theta = s(n)$. By the second equation, $s(n) + \theta = s(n + \theta)$, and by the induction’s hypothesis, $s(n + \theta) = s(n)$. Done.

Case 2: Assume that $m + N = N + m$ for some $m$ and for any $N$, and prove that $s(m) + N = N + s(m)$ for any $N$. By the second equation and the induction hypothesis, it follows that all what is left to prove is $N + s(m) = s(N + m)$, which we prove again by induction:

Case 2.1: $N = \theta$. The equality $\theta + s(m) = s(\theta + m)$ is immediate because of the first equation.

Case 2.2: Assume that $n + s(m) = s(n + m)$ for some $n$ and show $s(n) + s(m) = s(s(n) + m)$. Indeed, $s(n) + s(m) = s(n + s(m)) = s(s(n + m))$ by the second equation and the induction hypothesis, and $s(s(n) + m) = s(s(n + m))$ by the second equation.

Therefore, by several applications of induction we proved that addition on Peano natural numbers is commutative. Hence, one would naturally expect that addition should be commutative for any implementation of natural numbers satisfying the Peano axioms in PEANO-NAT. Well, let us consider the following implementation $S$:

- Natural numbers are interpreted as strings;
- $\theta$ is interpreted as the empty string;
- $s(N)$ is the string $aN$, that is, the character $a$ concatenated with the string $N$;
- $+_{\text{string}}$ is implemented as string concatenation.

$S$ obviously satisfies the two axioms of PEANO-NAT. However, addition is not commutative in $S$! Where is the problem? What’s going on here? There is nothing wrong, just that

*Proofs by induction are NOT valid for all possible models/implementations, but only for special ones!*

At this stage, one can admittedly say: well, but in the string model you can only build natural numbers by adding an ‘a’ in front of a string, so you can only get strings of a’s, in which case addition is commutative. That means that one actually had in mind a special model all the time, namely one in which all the numbers
need to be reached starting with zero and applying successor operations. For this model, induction is indeed a valid proof principle, because this model is the initial one. But note that there is no axiom in the Peano definition of natural numbers saying that in any model natural numbers can only be obtained starting with a zero and applying a finite number of successor operations. That was only implicit in our mind, but not explicit in the axioms.

One thing to keep in mind when we talk about specifications is that they give us a set of constraints that the possible acceptable worlds should satisfy. They do not place us in a unique world. Also, in the context of programming language design, we do not want to restrict too much the class of implementations by design! One implementation of a programming language can be by using a compiler, another by using an interpreter, yet another by translating it into a different programming language.

When one proves a property about a specification, that property is expected to hold in all possible worlds. However, our major points above are:

1. Properties one proves by induction are not valid in all possible worlds, so one has to restrict the possible universes to those in which induction is valid! Those are called "reachable" in the literature, but this goes beyond the scope of the class. What’s important is that the initial model is reachable; that’s the model you should have in mind when you write executable language specifications.

2. Once one accepts the restricted set of possible universes in which induction is valid, then one can also have a methodological clean way to develop specifications: think of what are the constructors that you would use in doing proofs by induction, and then make sure you define your new operations for each of those.

From here on, we will consider only specifications whose intended models are initial, so proofs by induction are valid.

**Exercise 21.** Show that addition is associative in PEANO-NAT and that multiplication is commutative and associative in PEANO-NAT*, where we also replace mult by its mix-fix variant _*_. These proofs need to be done by induction. Describe also a model/implementation of PEANO-NAT*, where multiplication is implemented in such a way that it is neither commutative nor associative. You can extend the one on strings if you wish.

### 2.4.6 Defining Data-Types Equationally

also known as algebraic data-types

- Lists
- Bags
- Sets
- Maps

Make it clear that we define Partial Finite-Domain Maps

42
sort:  
   List
subsort:  
   A < List
operations:  
   nil  :  \rightarrow  List
   _ , _  :  List \times List \rightarrow List
equations:  
   nil, L = L
   L, nil = L
   (L_1, L_2), L_3 = L_1, (L_2, L_3)

Figure 2.4: Lists of elements on sort A defined as an algebraic datatype. Variables L,L_1,L_2,L_3 have sort List.

sort:  
   Bag
subsort:  
   A < Bag
operations:  
   \emptyset  :  \rightarrow Bag
   _ , _  :  Bag \times Bag \rightarrow Bag
equations:  
   \emptyset, B = B
   B_1, B_2 = B_2, B_1
   (B_1, B_2), B_3 = B_1, (B_2, B_3)

Figure 2.5: Bags of elements on sort A defined as an algebraic datatype. Variables B,B_1,B_2,B_3 have sort Bag.

sort:  
   Set
subsort:  
   A < Set
operations:  
   \emptyset  :  \rightarrow Set
   _ , _  :  Set \times Set \rightarrow Set
equations:  
   X, X = X
   \emptyset, S = S
   S_1, S_2 = S_2, S_1
   (S_1, S_2), S_3 = S_1, (S_2, S_3)

Figure 2.6: Sets of elements on sort A defined as an algebraic datatype. Variables S,S_1,S_2,S_3 have sort Set, and X has sort A.
say that general, possibly infinite-domain partial functions, like any other non-r.e. domains, cannot be defined using any computationally-based technique; one can still define certain infinite structures, in particular partial functions, provided that one has finite presentations of them, such as recursive definitions.

define the update to only work on states that are defined in the variable to update.
allow \( \sigma[\perp/x] \).
also, add a defined Boolean operation which checks whether a partial function is defined in a certain element.
the above are needed for consistency with the conventional executable semantics.

Partial finite-domain functions can be easily defined equationally as an algebraic data-type of sets of pairs, more precisely as sets of bindings from elements in their sources to elements in their targets, respectively. This works because partial finite-domain functions, when regarded as binary relations, are indeed finite sets of pairs. Following the presentation of partial functions in Section 2.1.2, we also define lookup and update operations on partial function. We show that well-formed structures defined this way indeed correspond to partial finite-domain functions, so they can be regarded as such partial functions for all practical purposes from now on.

Figure 2.7 shows our definition of partial finite-domain functions as an algebraic data-type. We define a sort \( \text{PFun} \) together with a pairing “binding” constructor \( _\mapsto : A \times B \to \text{PFun} \) for it and together with an associative and commutative union construct \( _\cup : \text{PFun} \times \text{PFun} \to \text{PFun} \) with \( \emptyset \) as its identity to put together such bindings. The equational definitions of operations \( _\mapsto : \text{PFun} \times A \to B \) and \( _/[\_] : \text{PFun} \times B \times A \to \text{PFun} \) are given by equations that operate modulo the associativity and commutativity of \( _\mapsto, _\cup \) but it could also be given many different ways, including ones that do not make use of AC matching.

Note the \( a \neq a' \) appearing as a condition is not a negative condition, but rather a Boolean predicate, which can be equationally defined for any constructor-based type such as the type of integers or of variables, for example. Such “built-ins” appearing in these definitions and others in this book are easily definable as initial models of corresponding equational theories. And indeed, when performing formal proofs, one will make use of these equational definitions of the built-ins.

Definition 11. Let \( T \) be the initial algebra of the equational specification in Figure 2.7. A term \( t \in T_{\text{PFun}} \) is well-formed as a partial functions, or simply well-formed, if and only if \( \text{Partial-Function}(A,B): t = (a_1 \mapsto b_1, \ldots, a_n \mapsto b_n) \) for some \( a_i \in A \) and \( b_i \in B \) for each \( i \in \{1, \ldots, n\} \), such that \( a_i \neq a_j \) for any \( i \neq j \). Let \( T_{\text{PFun}}^\text{wf} \subset T_{\text{PFun}} \) denote the set of well-formed terms.

In other words, a term of sort \( \text{PFun} \) is well-formed if and only if, when regarded as a binary relation, it corresponds to a partial function.

Proposition 9. The following hold:
1. \( \emptyset \in T_{\text{PFun}}^\text{wf} \);
2. If \( t \in T_{\text{PFun}}^\text{wf}, a \in A, b \in B, \) then \( t[b/a] \in T_{\text{PFun}}^\text{wf} \).
Partial-Function(A,B)

sorts:
PFun

operations:
− → : A × B → PFun
∅ : → PFun
− , − : PFun × PFun → PFun
− (, ) : PFun × A → B
− [ , ] : PFun × B × A → PFun

equations:
(F1, F2), F3 = F1, (F2, F3)
F1, F2 = F2, F1
∅, F = F
(F, a ↦ b)(a) = b
(F, a ↦ b)[b′/a] = (F, a ↦ b′)
(F, a ↦ b)[b′/a′] = (F[b′/a′], a ↦ b) if a ≠ a′
∅[b/a] = a ↦ b

Figure 2.7: Partial finite-domain functions defined as an algebraic datatype. We assume the definition of the sorts A and B already given. The variables a,a′ have the sort A, the variables b,b′ have sort B, and F,F1,F2,F3 have sort PFun.

3. If (t1, t2) ∈ T_{WF}^{PFun} then t1 ∈ T_{WF}^{PFun} and t2 ∈ T_{WF}^{PFun}.

Thus, if one always starts with the term ∅ and then only uses the provided update operation to construct terms of sort PFun, then one is guaranteed that one only obtains well-formed terms. Moreover, any subterm of a well-formed term is also well-formed.


We next show that well-formed terms in T_{WF}^{PFun} and partial finite-domain functions in [A → B]^finite are equivalent. We show it by defining isomorphic mappings between the two:

Definition 12. Let us consider two functions
- φ : T_{WF}^{PFun} → [A → B]^finite and
- ψ : [A → B]^finite → T_{WF}^{PFun}

defined as follows:
- for any t ∈ T_{WF}^{PFun}, a ∈ A and b ∈ B, φ(t) is defined in a and φ(t)(a) = b if and only if
  PARTIAL-FUNCTION(A,B) ⊢ t = (t′, a ↦ b) for some t′ ∈ T_{WF}^{PFun} and
- for any f : A → B of finite domain {a1, a2, . . . , an}, ψ(f) is defined as the term (a1 ↦ b1, a2 ↦ b2, . . . , an ↦ bn), where bk = f(ak) for each k ∈ {1, 2, . . . , n}.

As expected, the two mappings above are well-defined, are inverse to each other, and commute with the state lookup and update operations:
Proposition 10. With the notation in Definition 12, the mappings $\varphi$ and $\psi$ are well-defined modulo the equations in $\text{Partial-Function}(A,B)$, that is:

(\varphi \text{ well-defined}) If $t_1, t_2 \in T_{\text{PFun}}^\text{w}$ such that $\text{Partial-Function}(A,B) \vdash t_1 = t_2$, then $\varphi(t_1) = \varphi(t_2)$;

(\psi \text{ well-defined}) If $f : A \to B$ is a partial function of finite domain $\{a_1, a_2, \ldots, a_n\}$ with $f(a_k) = b_k$ for each $k \in \{1, 2, \ldots, n\}$ and if $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ is a permutation of the numbers $\{1, 2, \ldots, n\}$, then $\text{Partial-Function}(A,B) \vdash (a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_n \mapsto b_n) = (a_{\pi(1)} \mapsto b_{\pi(1)}, a_{\pi(2)} \mapsto b_{\pi(2)}, \ldots, a_{\pi(n)} \mapsto b_{\pi(n)})$.

Moreover, the following hold for any $f \in [A \to B]^\text{finite}$ and for any $t \in T_{\text{PFun}}^\text{w}$:

1. $\varphi(\psi(f)) = f$;
2. $\text{Partial-Function}(A,B) \vdash \psi(\varphi(t)) = t$;
3. For any $a \in A$ and any $b \in B$, the following hold:
   - $f(a) = b$ iff $\text{Partial-Function}(A,B) \vdash \psi(f)(a) = b$, and
   - $\text{Partial-Function}(A,B) \vdash t(a) = b$ iff $\varphi(t)(a) = b$;
4. For any $a \in A$ and any $b \in B$, the following hold:
   - $\text{Partial-Function}(A,B) \vdash \psi(f[b/a]) = \psi(f)[b/a]$;
   - $\varphi(\psi(f)[b/a]) = \varphi(f)[b/a]$.


the following material comes from state definitions and may not be needed anymore

We can now naturally define the

Definition 13. A state $\sigma$ is equivalent to a finite partial function $\sigma : \text{Var} \to \text{Int}$, written $\sigma$ = $\sigma$, if $\sigma$ is well-formed and for any $x \in \text{Var}$ and $i \in \text{Int}$, $\sigma(x) = i$ iff $\text{STATE} \vdash \sigma[\{x\}] = i$.

In other words, $\sigma$ = $\sigma$ if and only if $\sigma$, together with equational reasoning using the definition STATE, behaves like $\sigma$.

The proposition below states that, for all practical reasons, our equational definition of the state is completely equivalent to the one defining states as finite partial functions.

Proposition 11. Let $x, x' \in \text{Var}$, $i, i' \in \text{Int}$, $\sigma, \sigma' \in \text{State}$ and finite partial functions $\sigma, \sigma' : \text{Var} \to \text{Int}$. Then the following hold:

1. $\text{empty} = \emptyset$ where $\emptyset$ is the function undefined everywhere.
2. $(\sigma, x \mapsto i) \approx \sigma$ implies that $\sigma = \sigma \setminus \{x\}$, where $\sigma \setminus \{x\}$ is defined as $\sigma$ restricted to $\text{Dom}(\sigma) \setminus \{x\}$.
3. If $\sigma \approx \sigma$ then also $\sigma[x \leftarrow i] \approx \sigma[x \leftarrow i]$.

Proof. 1. Trivial, since there is no $x \in \text{Var}$ and $i \in \text{Int}$ such that $\text{STATE} \vdash \text{empty}[x] = i$. 46
2. Suppose that \((\sigma, x \mapsto i) \simeq \sigma\). Let \(\sigma'\) be such that \(\sigma \simeq \sigma'\). We prove that \(\text{Dom}(\sigma') = \text{Dom}(\sigma) \setminus \{x\}\) and \(\sigma'(x) = \sigma(x)\) for any \(x' \in \text{Dom}(\sigma')\). Consider an arbitrary \(x' \in \text{Var}\). If \(x' = x\) then there is no \(i' \in \text{Int}\) such that \(\text{State} \vdash \sigma(x) = i'\), since otherwise we would have \(\text{State} \vdash \sigma = (\sigma', x \mapsto i')\) for some state \(\sigma'\), which contradicts the well-formedness of \((\sigma, x \mapsto i)\); therefore, \(\sigma'\) is not defined on \(x\). If \(x' \neq x\), then \(\text{State} \vdash \sigma(x') = \sigma'(x')\), therefore \(\sigma'\) is defined on \(x'\) iff \(\sigma\) is defined on \(x'\), and, if so, then \(\sigma'(x') = \sigma(x')\).

3. Suppose \(\sigma \simeq \sigma\). We distinguish two cases—if \(\sigma\) is defined on \(x\) or if \(\sigma\) is not defined on \(x\). Let us first consider the case where \(\sigma\) is defined on \(x\). Let us say that \(\sigma(x) = i'\) for some \(i' \in \text{Int}\). Then it must be the case that \(\text{State} \vdash \sigma(x) = i'\), which can only happen if \(\text{State} \vdash \sigma = (\sigma', x \mapsto i')\) for some \(\sigma'\in \text{State}\), so \(\text{State} \vdash \sigma(x \leftarrow i) = (\sigma', x \mapsto i)\). We next show that \(\sigma(x \leftarrow i) \simeq \sigma(x \leftarrow i)\). Let \(x'\) be an arbitrary variable in \(\text{Var}\); there are two cases to analyze, when \(x' = x\) and when \(x' \neq x\). If \(x' = x\) then

\[
\text{State} \vdash (\sigma(x \leftarrow i))[x'] = (\sigma', x \mapsto i)[x'] = i
\]

and note also that \(\sigma(x \leftarrow i)[x'] = i\). If \(x' \neq x\) then

\[
\begin{align*}
\text{State} & \vdash (\sigma(x \leftarrow i))[x'] \\
& = (\sigma', x \mapsto i)[x'] \\
& = \sigma'[x'] \\
& = (\sigma', x \mapsto i')[x'] \\
& = \sigma[x']
\end{align*}
\]

and note also that, since \(\sigma \simeq \sigma\), we have \(\text{State} \vdash \sigma(x') = j\) for some \(j \in \text{Int}\) iff \(\sigma'[x'] = j\), which, since \(x' \neq x\), happens iff \(\sigma(x \leftarrow i)[x'] = j\). We have therefore shown that \(\sigma(x \leftarrow i) \simeq \sigma(x \leftarrow i)\) whenever \(\sigma\) is defined in \(x\).

If \(\sigma\) is not defined in \(x\), it means that there is no \(j \in \text{Int}\) such that \(\text{State} \vdash \sigma(x) = j\), so there is no \(j \in \text{Int}\) and \(\sigma' \in \text{State}\) such that \(\text{State} \vdash \sigma = (\sigma', x \mapsto j)\). If \(\text{State} \vdash \sigma = \text{empty}\) then we are done, since \(\text{State} \vdash (x \mapsto i)[x'] = i'\) iff \(x = x'\) and \(i = i'\). If it is not the case that \(\text{State} \vdash \sigma = \text{empty}\), it must be that \(\text{State} \vdash \sigma = (x_1 \mapsto i_1, \ldots, x_n \mapsto i_n)\) with \(x_k \neq x\) for all \(k \in \{1, \ldots, n\}\). This leads to \(\text{State} \vdash \sigma(x \leftarrow i) = \cdots = (x_1 \mapsto i_1, \ldots, x_l \mapsto i_l)[x \leftarrow i](x_{k+1} \mapsto i_{k+1}, \ldots, x_n \mapsto i_n) = \cdots = (\text{empty}[x \leftarrow i], \sigma) = (x \mapsto i, \sigma) = (\sigma, x \mapsto i)\). Thus \(\text{State} \vdash \sigma(x \leftarrow i)[x'] = i'\) iff either \(x' \neq x\) and \(\text{State} \vdash \sigma(x') = i'\) or \(x' = x\) and \(i' = i\) iff either \(\sigma(x) = i'\) or \(\sigma'[x] = i'\).

Because of the proposition above, we take the liberty to make no distinction between states represented as finite partial functions and states represented as equationally defined finite sets of bindings as above, and, for simplicity, we use the symbols \(\sigma, \sigma', \sigma_1, \text{etc.}\), to range over states in either of the two representations.

---

The above material comes from state definitions and may not be needed anymore.
sort:
Stream
operations:
  _ : _ : Int × Stream → Stream
head : Stream → Int
tail : Stream → Stream
zeros : → Stream
zip : Stream × Stream → Stream
add : Stream → Stream
fibonacci : → Stream
equations:
  head(X : S) = X
  tail(X : S) = S
  zeros = 0 : zeros
  zip(X : S1, S2) = X : zip(S2, S1)
  add(X1 : X2 : S) = (X1 +_{Int} X2) : add(S)
  fibonacci = 0 : 1 : add(zip(fibonacci, tail(fibonacci)))

Figure 2.8: Streams of integers defined as an algebraic datatype. The variables S, S1, S2 have sort Stream and the variables X, X1, X2 have sort Int.

Streams

Figure 2.8 shows an example of a data-structure whose elements are infinite sequences, called streams, together with several particular streams and operations on them. Here we prefer to be more specific than in the previous examples and work with streams of integers. We assume the integers and operations on them already defined; specifically, we assume Int to be their sort and operations on them indexed with Int to distinguish them from other homonymous operations, e.g., +_{Int}, etc. The operation _ : _ adds a given integer to the beginning of a given stream, and the dual operations head and tail extract the head (integer) and the tail (stream) from a stream. The stream zeros contains only 0 elements. The stream operation zip merges two streams by interleaving their elements, and add generates a new stream by adding any two consecutive elements of a given stream. The stream fibonacci consists of the Fibonacci sequence (see Exercise 25).

It is interesting to note that the equational specification of streams in Figure 2.8 is one where its initial algebra semantics is likely not the model that we want. Indeed, the initial algebra here would consists of infinite classes of finite terms, where any two terms in any class are provably equal, for example {zeros, 0 : zeros, 0 : 0 : zeros, ...}. While this is a valid and interesting model of streams, it is likely not what one has in mind when one thinks of streams as infinite sequences. Nevertheless, the intended stream model is among the models/algebras of this equational specication, so any equational deduction or reduction that we perform, with or without strategies, is sound (see Exercise 26).
2.4.7 Notes

Equational encodings of general computation into equational deduction are well-known; for example, \cite{7,1} show such encodings, where the resulting equational specifications, if regarded as term rewrite systems (TRSs), are confluent and terminate whenever the original computation terminates. Our goal in this section is to discuss equational encodings of (Turing machine) computation. These encodings will be used later in the paper to show the \(\Pi_2^0\)-hardness of the equational satisfaction problem in the initial algebra. While we could have used existing encodings of Turing machines as TRSs, however, we found them more complex and intricate for our purpose in this paper than needed. Consequently (and also for the sake of self-containment), we recall the more recent (simple) encoding and corresponding proofs from \cite{65}. Since the subsequent encoding is general purpose rather than specific to our \(\Pi_2^0\)-hardness result, the content of this section may have a more pedagogical than technical nature. For example, the references to TRSs are technically only needed to prove the equational encoding correct, so they could have been removed from the main text and added only in the proofs, but we find them pedagogically interesting and potentially useful for other purposes. The equational encodings that follow can be faithfully used as TRS Turing-complete computational engines, because each rewrite step corresponds to precisely one computation step in the Turing machine; in other words, there are no artificial rewrite steps.

2.4.8 Exercises

Exercise 24. Eliminate the two equations in Figure 2.3 as discussed right after Theorem 8 and prove a result similar to Theorem 8 for the new representation.

Exercise 25. Show that the fibonacci stream defined in Figure 2.8 indeed defines the sequence of Fibonacci numbers. This exercise has two parts: first formally state what to prove, and second prove it.

Exercise 26. Consider the equational specification of streams in Figure 2.8. Define the intended model/algebra of streams over integer numbers with constant streams and functions on streams corresponding to the various operations in this specification. Then show that this model indeed satisfies all the equations in Figure 2.8. Describe also its default initial model and compare it with the intended model. Are they isomorphic?
2.5 Rewrite logic

\[ \text{define } \rightarrow^1 \]

define or explain RL’s concurrent rewriting

discuss how RL, by its AC rewriting and non-sharing approach, forces interleaving in situations where one would not want interleaving: \( a^* \rightarrow b^* \), etc.

we use lots of Bool conditions in our subsequent semantics. explain how these relate to the RL approach where one only uses equations or rules in conditions.

cite multi-sorted signatures, instead of redefining everywhere. add them in the first, background section, before we even talk about term rewriting.

move List\{Sort\}, etc., in the appropriate section under eq logic, and cite

Talk about confluence (for example, Theorem \[23\] needs it). Talk also about confluence criteria, such as orthogonality; needed for example to show the confluence of combinatory logic.

2.5.1 Equations versus Rewrite Rules

In contrast to term rewriting, which is just a method of computation, rewrite logic is a computational logic proposed by Meseguer \[42\] as a unified logic for (true) concurrency, which builds upon equational logic by extending it with rewrite rules. In equational logic, a number of sorts (types) and equations are defined, specifying which terms are equal. Equal terms are members of the same equivalence class. In other words, equal terms are regarded as being identical; in particular, equations can be soundly applied either from left-to-right or from right-to-left when reasoning about equational or rewrite logic theories. Rewrite logic adds rules to equational logic, thought of as irreversible transitions: a rewrite theory is an equational theory extended with rewrite rules, where equations can be applied in any direction, while rules can only be applied from left-to-right. In other words, a rewrite logic theory consists of:

- **Syntax**, consisting of sorts (syntactic categories) and operators on them, that can be used to define well-formed uninterpreted terms; as already mentioned, we prefer to use equivalent CFGs to define syntax in this paper;

- **Equations**, defining structural identities on terms: equal terms can be substituted for each other unrestricted anywhere. Intuitively, equal terms are the same thing represented differently, the same way \( \pi, 3.1459 \ldots \) or the circumference of a circle of diameter 1 are the same thing; and
• **Rewrite rules**, defining irreversible transitions between terms.

Here we define languages and language features as rewrite logic theories: the syntax of the rewrite logic theory captures the syntax of the language possibly extended with auxiliary operators needed for semantics, equations capture structural rearrangements or equivalences of the program configuration of computation structures and thus carry no computational meaning, and rules capture the intended computational steps and can apply in parallel (provided they don’t overlap).

Both equations and rules can be parametric in rewrite logic, in the sense that they can contain variables that act as placeholders for subterms that match them. To avoid defining the details of rewrite logic, including substitutions and matching, by a parametric equation or rule we mean a recursively enumerable set of equations or rules, one per each parameter instance that satisfies the side conditions. In other words, equations and rules in this paper are regarded as schematas (when executing them on rewrite engines, one may need to capture side conditions using conditions in rules and/or equations). Unlike in reduction semantics with evaluation contexts [25], there are no context restrictions on applications of equations and/or rules (extensions of rewrite logic have been proposed recently allowing variants of context-sensitivity, but we are not going to use those). This allows for rewrite logic to serve as a very simple and elegant foundation for concurrency, because rules and equations can indeed be applied concurrently. Rewrite logic admits a complete proof system and an initial model semantics [42] that makes inductive proofs rigorous and valid. The intuition for why rewrite logic is a unified logic for concurrency comes from two observations: (1) the rewriting process is inherently parallel, in the sense that non-overlapping parts of a term can be rewritten concurrently, and thus fits well with current trends in concurrent system architecture; and (2) as shown in [42, 36], many other models of concurrent computation can be represented within rewrite logic, capturing faithfully the intended true or interleaved concurrency of each model.

Rewrite logic is connected to term rewriting in that the latter can be used to execute theories in the former. Indeed, most of the equations \( l = r \) can be transformed into term rewriting rules \( l \rightarrow r \), thus providing a means to taking a rewrite logic theory, together with an initial term, and executing it using the underlying TRS. Some of the TRS steps therefore correspond to equations, so can be viewed as structural transformations rather than computational steps. Any of the existing rewrite engines can be used for this purpose. It is important, though, not to confuse orientation of equations for executability purposes with rewrite rules in rewrite logic! As mentioned above, the actual semantics of a rewrite logic theory is that rewrite rules can apply concurrently on equivalence classes of terms obtained modulo all the equations! Not all equations can be oriented into rewrite rules for execution purposes; for example, an equation stating commutativity (C) of a binary operator cannot be regarded as a rewrite rule, because it would lead to non-termination; similarly, equations for associativity (A) or even ones defining identities, or units, of binary operators (I) may not always be desired to be regarded as (irreversible) rewrite rules. For that reason, some rewrite engines provide support for rewriting modulo special equations such as A, C, I.

An associative binary operator on a sort \( S \) can also be regarded as a list construct for \( S \), an ACI one as a list construct; to obtain a set, one also needs to add an explicit idempotency equation in addition to the ACI equations. Since lists and (multi-)sets admit straightforward equational definitions in rewrite logic, from here on in this paper, we assume lists and (multi-)sets over any sort whenever needed; for a sort \( \text{Sort} \), \( \text{List}\{\text{Sort}\} \) denotes comma-separated lists of terms of sort \( \text{Sort} \), and \( \text{Set}\{\text{Sort}\} \) denotes white space separated (multi-)sets of terms of sort \( \text{Sort} \). For all these structures, we use \( \sim \) as unit (nil, empty, etc.). If one wants a different list or set separator, then one writes it as a subscript, while if one wants a different unit then one writes it as a superscript. For clarity, we take the liberty to occasionally mention as subscripts and/or superscripts even the default constructs and units for lists and sets. For example, \( \text{List}\sim\{K\} \) stays for \( \sim \)-separated lists of terms of sort \( K \). We also assume, whenever...
needed, product sorts; these are written conventionally \((S_1 \times S_2)\) and are also easy to define in rewrite logic.

We next show some examples of rewrite theories modulo equations (modulo A and AI, and modulo AC and ACI). First, let us consider two simple examples, showing also how intuitively rewrite logic and concurrency fit together. Suppose that one would like to define (comma-separated) sorted lists of integers. Then all one needs to do is to add the following bubble sort rule (to the AI equations for lists):

\[
i, j \to j, i \quad \text{when } i > j
\]

Here is an example of a nine-step rewrite logic derivation in the one-rule rewrite theory above (underlined subterms are redexes where the rule applies in the corresponding step):

\[
\begin{align*}
7, 6, 5, 4, 3, 2, 1 & \to 6, 7, 4, 5, 3, 1, 2 \\
3, 4, 6, 1, 7, 2, 5 & \to 3, 4, 1, 6, 2, 7, 5 \\
& \to 3, 1, 4, 2, 6, 5, 7 \\
& \to 1, 3, 2, 4, 5, 6, 7 \\
& \to 1, 2, 3, 4, 5, 6, 7
\end{align*}
\]

Note that the rewrite rule is allowed to apply concurrently, but that it is not enforced to apply everywhere it can; for example, the rule was purposely not applied on the subterm “5, 1” in the third step above. Also, note that many applications of the associativity equation for \(\_ \_\) were necessary in order to structurally change the term so that the rewrite rule could match and apply (sometimes multiple times). Note also that the applications of rules is non-deterministic and that one rule application choice may disable one or more other potential rule applications; for example, in the first step above one could have chosen the redex “6, 5”, which would have disabled the redexes “7, 6” and “5, 4”. This “disabling” is, of course, just an informal way of saying that equations (such as associativity) can be applied many different and sometimes exclusive ways in order for the rules to match. It is obvious that the resulting rewrite system terminates (modulo AI), because the number of misplaced numbers decreases at each application of the rule, concurrent or not. It is also obvious that the normal-form lists are sorted, so the one-rule rewrite system above gives a correct sorting algorithm. What may be less obvious is that, when executed on a parallel rewrite engine with plenty of cores, this trivial rewrite system gives a parallel algorithm that can sort in worst-case linear time.

Consider now a simple game. A group of children get a large bag (an ACI operator, giving a multiset) with black and white balls, as well as arbitrarily many additional balls. Each child can remove two balls from the bag, whenever they want and concurrently, but then immediately put back in the bag a black ball if both extracted balls had the same color, or a white ball if they had different colors. Other children may dynamically join the game. We can define this problem as a rewrite theory as follows \((b\) ranges over \(Ball\)):

\[
\begin{align*}
Ball & ::= \circ | \bullet \\
Bag & ::= \text{Set}(Ball) \\
\circ \bullet & \to \circ \\
b \bullet & \to \bullet \quad \text{where } b \in \{\circ, \bullet\}
\end{align*}
\]

It is obvious that as far as children keep playing the game it will eventually terminate, because at each extraction the number of balls decreases by at least one. The rules above can apply concurrently as far as there are at least two balls in the bag. What is not immediately obvious is that, despite the high degree of parallelism in this ACI rewrite system, it is actually confluent: the parity of the white balls does not change.

Let us now consider an extension of the simple calculator language above with “,”-separated lists and with “;”-separated lists of lists (let us not worry about typing such programs for now) and with a permutation construct taking a number \(n\) and generating the list of permutation lists over the elements \(\{1, 2, \ldots, n\}\) (here \(n, m\) range over natural numbers, \(p\) over “,”-separated lists of natural numbers, and \(pl\) over “;”-separated lists

52
We here did not bother to produce an error if the map was not defined in
while ML and Scheme took 83 and 92 seconds, respectively. None of these systems were able to calculate
Nat \times \text{equations} as schematas, so they are allowed to have side conditions.
only unconditional rules and equations, though we adopt common conventions in logics and regard rules and
mentioned above, in the theoretical developments of rewrite logic semantics and K in this paper we assume
define the update operation; the first uses again ACI matching, while the second has a side condition. As
kind Maude one would define it to return an element of \text{Exp}. The remaining two rules
define the update operation; the first uses again ACI matching, while the second has a side condition. As
one can show that maps contain no duplicate pairs. The first rule shows a first example of ACI matching:
applications of associativity and commutativity equations may take place in the background to bring the
note that \text{Nat} \times \text{Exp}. The equation assures that maps are indeed sets, not multi-sets (note that this set requirement is
be thought of as a degenerated "","-separated list of ",-separated lists; indeed, thanks to matching modulo
identified (I), the fourth rule also applies when \text{p} is \cdot \text{(since } \text{m} \text{p} \text{ matches } 1 \text{ with } \text{m} = 1 \text{ and } \text{p} = \cdot \text{). In our experiments, the rewrite theory above, when executed in Maude, outperformed best implementations
-efficient implementations, in addition to their conciseness and mathematical rigor.
To further exemplify the use of ACI operators and corresponding matching, let us now consider an
extension of the simple calculator language above with partial maps (from natural numbers to expressions)
defined by their graphs (i.e., as sets of pairs \(\langle n, e \rangle\)), as well as update and lookup constructs:
In the above, we assumed white-space-separated sets and the notation \([n, e]\) for pairs in the product sort
\text{Nat} \times \text{Exp}. The equation assures that maps are indeed sets, not multi-sets (note that this set requirement is
not really necessary in this case, because if one constructs maps only with the provided update interface then
one can show that maps contain no duplicate pairs. The first rule shows a first example of ACI matching:
\text{e e} = \text{e}
\langle[n, e] \text{ g}\rangle\langle n \rangle \rightarrow \text{e}
\langle[n, e] \text{ g}\rangle\langle e' / n \rangle \rightarrow \langle n, e'\rangle \text{ g}
g\langle e' / n \rangle \rightarrow \langle n, e'\rangle \text{ g} \quad \text{when } g \text{ contains no pair } \langle n, -\rangle
In theory, many applications of associativity and commutativity equations may take place in the background to bring the
pair \([n, e]\) on the first position in the set. In practice, ACI rewrite engines like Maude implement efficient
data-structures and indexing to perform such matches in essentially logarithmic time (in the size of the set).
We here did not bother to produce an error if the map was not defined in \text{n}; if \text{g} has no pair of the form \([n, -]\),
then the term \text{g}\langle n \rangle \text{ is stuck and thus can be regarded as a "core dump" error message. On rewrite engines
providing support for errors, one can define lookup as a potentially error generating operator; for example, in
Maude one would define it to return an element of \text{kind } \text{Exp}\text{ instead of sort } \text{Exp}. The remaining two rules
define the update operation; the first uses again ACI matching, while the second has a side condition. As
mentioned above, in the theoretical developments of rewrite logic semantics and K in this paper we assume
only unconditional rules and equations, though we adopt common conventions in logics and regard rules and
equations as schematas, so they are allowed to have side conditions.
The simple calculator language above, with or without its extensions, is rather trivial and therefore cannot be used as an argument that rewrite logic can serve as a solid foundation for programming languages. Indeed, the main problem when defining non-trivial languages is that the applications of reductions needs to be somehow controlled, to ensure the correct evaluation flow in a program, and, moreover, applications of rules may need data that is not available locally, such as, e.g., values associated to locations in a shared, top-level store. Rewrite logic semantics (RLS), proposed by Meseguer and Rosu, builds upon the strong belief, supported by extensive empirical evidence, that arbitrarily complex programming languages can, in fact, be easily defined as rewrite logic theories. By doing so, one gets essentially for free not only an interpreter and an initial model semantics for the defined language, but also a series of formal analysis tools obtained as instances of existing tools for rewriting logic.

As mentioned above, operationally speaking the major difference between conventional reduction semantics, with or without evaluation contexts, and rewrite logic semantics is that the former typically impose contextual restrictions on applications of reduction steps and the reduction steps happen one at a time, while the latter imposes no such restrictions. To avoid undesired applications of rewrite steps, one has to follow certain techniques and obey certain methodologies when using rewrite logic. In particular, as shown in [74], the more conventional language definitional styles (small-step and big-step SOS, reduction semantics with evaluation contexts, chemical abstract machines, continuation-based semantics, etc.) can be faithfully captured in rewrite logic by appropriate particular uses of its general reduction machinery. Consequently, one can define a language many different ways in rewrite logic, each with its advantages and disadvantages; indeed, the fact that one faithfully captures a small-step SOS definition as a rewrite logic theory does not mean that the resulting rewrite theory will give a true concurrency semantics to the original SOS definition: the adverb “faithfully”, justified by theorems proved in [74], ensures that the corresponding rewrite theories have exactly the same strengths and limitations as the original definitions.

In this paper, we discuss the K rewrite logic technique, which was first introduced in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign in Fall 2003, as a means to define executable concurrent languages in rewrite logic using the Maude executable specification system. K and its corresponding Maude tool support incrementally improved every year since then, as seen in the series of technical reports. The latest of these reports also contains a more formal and detailed algebraic description of K. K is reminiscent of abstract state machines and continuations, and glosses over language-irrelevant rewrite logic details. Analyzing the various faithful translations of language definitional styles into rewrite logic discussed in detail in [74], as well as their advantages and disadvantages when regarded as rewrite logic theories, we confidently believe that K captures and reflects best the strengths of rewrite logic as a semantic framework for programming languages and related features.

---

A Boolean term \( b \) that appears as a condition is meant to be an equality \( b = \text{true} \)

---

From I&c

Rewrite logic is a computational logic that can be efficiently implemented and that has good properties as a general and flexible logical and semantic framework, in which a wide range of logics and models of computation can be faithfully represented. In particular, for programming language semantics it provides the RLS framework, of which we emphasize the operational semantics aspects in this paper (for the mathematical aspects of RLS see [46][48]).
Two key points to explain are: (i) how rewrite logic combines equational logic and traditional term rewriting; and (ii) what the intuitive meaning of a rewrite theory is all about. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with $\Sigma$ a signature of function symbols, $E$ a set of (possibly conditional) $\Sigma$-equations, and $R$ a set of $\Sigma$-rewrite rules which in general may be conditional, with conditions involving both equations and rewrites. That is, a rule in $R$ can have the general form

$$(\forall X) \ t \rightarrow t' \text{ if } (\bigwedge_i u_i = u'_i) \land (\bigwedge_j w_j \rightarrow w'_j)$$

Alternatively, such a conditional rule could be displayed with an inference-rule-like notation as

$$\frac{(\bigwedge_i u_i = u'_i) \land (\bigwedge_j w_j \rightarrow w'_j)}{t \rightarrow t'}$$

Therefore, the logic’s atomic sentences are of two kinds: equations, and rewrite rules. Equational theories and traditional term rewriting systems then appear as special cases. An equational theory $(\Sigma, E)$ can be faithfully represented as the rewrite theory $(\Sigma, E, \emptyset)$; and a term rewriting system $(\Sigma, R)$ can likewise be faithfully represented as the rewrite theory $(\Sigma, \emptyset, R)$.

Of course, if the equations of an equational theory $(\Sigma, E)$ are *confluent*, there is another useful representation, namely, as the rewrite theory $(\Sigma, \emptyset, \overrightarrow{E})$, where $\overrightarrow{E}$ are the rewrite rules obtained by orienting the equations $E$ as rules from left to right. This representation is at the basis of much work in term rewriting, but by implicitly suggesting that rewrite rules are just an efficient technique for equational reasoning it can blind us to the fact that rewrite rules can have a much more general non-equational semantics. This is the whole *raison d’être* of rewrite logic. In rewrite logic a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ axiomatizes a concurrent system, whose states are elements of the algebraic data type axiomatized by $(\Sigma, E)$, that is, they are $E$-equivalence classes of ground $\Sigma$-terms, and whose atomic transitions are specified by the rules $R$. The inference system of rewrite logic described below then allows us to derive as proofs all the possible concurrent computations of the system axiomatized by $\mathcal{R}$, that is, concurrent computation and rewrite logic deduction coincide.

### 2.5.2 Proof System and Soundness

The inference rules below assume a typed setting, in which $(\Sigma, E)$ is a membership equational theory having sorts (denoted $s, s', s'',$ etc.), subsort inclusions, and kinds (denoted $k, k', k'',$ etc.), which gather together connected components of sorts. Kinds allow error terms like $3/0$, which has a kind but no sort. Similar inference rules can be given for untyped or simply typed (many-sorted) versions of the logic. Given $\mathcal{R} = (\Sigma, E, R)$, the sentences that $\mathcal{R}$ proves are universally quantified rewrites of the form $(\forall X) \ t \rightarrow t'$, with $t, t' \in T_\Sigma(X)_k$, for some kind $k$, which are obtained by finite application of the following *rules of deduction*:

- **Reflexivity.** For each $t \in T_\Sigma(X)$,
  $$(\forall X) \ t \rightarrow t$$

- **Equality.**
  $$(\forall X) \ u \rightarrow v \quad E \vdash (\forall X) u = u' \quad E \vdash (\forall X) v = v'$$
  $$(\forall X) \ u' \rightarrow v'$$

- **Congruence.** For each $f : s_1 \ldots s_n \rightarrow s$ in $\Sigma$, with $t_i \in T_\Sigma(X)_{s_i}$, $1 \leq i \leq n$, and with $t'_{j_1} \in T_\Sigma(X)_{s_{j_1}}$, $1 \leq l \leq m$,
  $$(\forall X) \ t_{j_1} \rightarrow t'_{j_1} \quad \ldots \quad (\forall X) \ t_{j_m} \rightarrow t'_{j_m}$$
  $$(\forall X) \ f(t_1, \ldots, t_{j_1}, \ldots, t_{j_m}, \ldots, t_n) \rightarrow f(t_1, \ldots, t'_{j_1}, \ldots, t'_{j_m}, \ldots, t_n)$$
• **Replacement.** For each \( \theta : X \rightarrow T_\Sigma(Y) \) and for each rule in \( R \) of the form

\[
(\forall X) \ t \rightarrow t' \text{ if } (\bigwedge_i u_i = u'_i) \land (\bigwedge_j w_j \rightarrow w'_j),
\]

\[
(\bigwedge_x(\forall Y) \ \theta(x) \rightarrow \theta'(x)) \land (\bigwedge_i \theta(u_i) = \theta(u'_i)) \land (\bigwedge_j \theta(w_j) \rightarrow \theta(w'_j)) \equiv (\forall Y) \ \theta(t) \rightarrow \theta'(t)
\]

where \( \theta' \) is the new substitution obtained from the original substitution \( \theta \) by some possibly complex rewriting of each \( \theta(x) \) to some \( \theta'(x) \) for each \( x \in X \).

• **Transitivity**

\[
(\forall X) \ t_1 \rightarrow t_2 \quad (\forall X) \ t_2 \rightarrow t_3 \quad \Rightarrow \quad (\forall X) \ t_1 \rightarrow t_3
\]

We can visualize the above inference rules as in Figure 2.9.

The notation \( \mathcal{R} \vdash t \rightarrow t' \) states that the sequent \( t \rightarrow t' \) is provable in the theory \( \mathcal{R} \) using the above inference rules. Intuitively, we should think of the inference rules as different ways of constructing all the (finitary) concurrent computations of the concurrent system specified by \( \mathcal{R} \). The **Reflexivity** rule says that for any state \( t \) there is an idle transition in which nothing changes. The **Equality** rule specifies that the states are in fact equivalence classes modulo the equations \( E \). The **Congruence** rule is a very general form of “sideways parallelism,” so that each operator \( f \) can be seen as a parallel state constructor, allowing its arguments to evolve in parallel. The **Replacement** rule supports a different form of parallelism, which could be called “parallelism under one’s feet,” since besides rewriting an instance of a rule’s left-hand-side to the corresponding right-hand-side instance, the state fragments in the substitution of the rule’s variables can also be rewritten. Finally, the **Transitivity** rule allows us to build longer concurrent computations by composing them sequentially.

A somewhat more general version of rewrite logic [FIXTHIS] allows rewrite theories of the form \( \mathcal{R} = (\Sigma, E \cup A, R, \phi) \), where the additional component \( \phi \) is a function assigning to each function symbol \( f \in \Sigma \) with \( n \) arguments a subset \( \phi(f) \subseteq \{1, \ldots, n\} \) of those argument positions that are frozen, that is, positions under which rewriting is forbidden. The above inference rules can then be slightly generalized. Specifically, the **Congruence** rule is restricted to non-frozen positions \( \{j_1, \ldots, j_m\} \), and the substitution \( \theta' \) in the **Replacement** rule should only differ from \( \theta \) for variables \( x \) in non-frozen positions. The generalized form \( \mathcal{R} = (\Sigma, E \cup A, R, \phi) \), makes possible a more expressive control of the possibility of rewriting under contexts already supported by the **Congruence** rule; that is, it endows rewrite theories with flexible context-sensitive rewriting capabilities\(^1\).

Note that, in general, a proof \( \mathcal{R} \vdash t \rightarrow t' \) does not represent an atomic step, but can represent a complex concurrent computation. In some of the mathematical proofs that we will give to relate different operational semantics definitions, it will be easier to work with a “one-step” rewrite relation \( \rightarrow^1 \), defined on ground terms. This relation is just the special case in which: (i) **Transitivity** is excluded; (ii) \( m = 1 \) in the **Congruence** rule (only one rewrite below); and (iii) **Replacement** is restricted, so that no rewriting of the substitution \( \theta \) to \( \theta' \) is allowed; and (iv) there is exactly one application of **Replacement**. The relation \( \rightarrow^{\leq 1} \) is defined by allowing either one or no applications of **Replacement** in the last condition. Similarly, one can define relations \( \rightarrow^{\geq 1} \).

---

\(^1\)We will not consider this general version. The interested reader is referred to [FIXTHIS]
(or \( \rightarrow^* \)) by controlling the number of applications of the Transitivity rule. However, it should be noted that rewrite logic does not have a built-in one-step rewrite relation, that being the reason for which we need a methodology to encode one-step-based formalisms such as SOS semantics. The one-step relation we define above is only at the deduction level and is introduced solely to help our proofs.

The whole point of RLS is then to define the semantics of a programming language \( \mathcal{L} \) as a rewrite theory \( \mathcal{R}_\mathcal{L} \). RLS uses the fact that rewrite logic deduction is performed modulo the equations in \( \mathcal{R}_\mathcal{L} \) to faithfully capture the desired granularity of a language's computations. This is achieved by making rewriting rules all intended computational steps, while using equations for convenient equivalent structural transformations of the state, or auxiliary “infrastructure” computations, which should not be regarded as computation steps. Note that this does not preclude performing also equational simplification with equations. That is, the set \( E \) of equations in a rewrite theory can often be fruitfully decomposed as a disjoint union \( E = E_0 \cup A \), where \( A \) is a set of structural axioms, such as associativity, commutativity and identity of some function symbols, and \( E_0 \) is a set of equations that are confluent and terminating modulo the axioms \( A \). A rewrite engine supporting rewriting modulo \( A \) will then execute both the equations \( E_0 \) and the rules \( R \) modulo \( A \) by rewriting. Under a condition called coherence \[\text{FIXTHIS}\] , this form of execution then provides a complete inference system for the given rewrite theory \((\Sigma, E, R)\). However, both conceptually and operationally, the execution of rules \( R \) and equations \( E_0 \) must be separated. Conceptually, what we are rewriting with \( R \) are \( E \)-equivalence classes, so that the \( E_0 \)-steps become invisible. Operationally, the execution of rules \( R \) and equations \( E_0 \) must be kept separate for soundness reasons. This is particularly apparent in the case of executing conditional equations and rules: for a conditional equation it would be unsound to use rules in \( R \) to evaluate its condition; and for a conditional rule it would likewise be unsound to use rules in \( R \) to evaluate the equational part of its condition.

There are many systems that either specifically implement term rewriting efficiently, so-called as rewrite engines, or support term rewriting as part of a more complex functionality. Any of these systems can be used as an underlying platform for execution and analysis of programming languages defined using the techniques proposed in this paper. Without attempting to be exhaustive, we here only mention (alphabetically) some engines that we are more familiar with, noting that many functional languages and theorem provers provide support for term rewriting as well: Asf+Sdf [81], CafeOBJ [21], Elan [10][11], Maude [18][17], OBJ [28], and Stratego [85]. Some of these engines can achieve remarkable speeds on today’s machines, in the order of tens of millions of rewrite steps per second.

---

lazy evaluation to be added in the next section; it is needed in Section

\[\text{FIXTHIS}\]
Theorem 10. The rewrite logic theory $\mathbf{R}_{M}^{\text{lazy}}$ is confluent. Moreover, the Turing machine $M$ and the rewrite theory $\mathbf{R}_{M}^{\text{lazy}}$ are step-for-step equivalent, that is,

$$\forall q_0, q', u, v, u', v' \in Q \cap \mathbb{B} \cup \{\bot\},\ 
(q, 0^{|u| + 1} \nu v 0^{|v|}) \rightarrow_M (q', 0^{|u'| + 1} u' \nu v' 0^{|v'|}) \text{ if and only if } \mathbf{R}_{M}^{\text{lazy}} \models q(\overrightarrow{u}, b : \overrightarrow{v}) \rightarrow q'(\overleftarrow{u'}, b' : \overrightarrow{v'}) \text{.}$$

2.5.3 Computation as Rewrite Logic Deduction

Building upon the equational representations of deterministic Turing machines in Section 2.4.3, here we show how we can associate rewrite theories to non-deterministic Turing machines so that there is a bijective correspondence between computational steps performed by the original Turing machine and rewrite steps in the corresponding rewrite theory. In non-deterministic Turing machines, the total transition function $M : (Q \times \{q_0\}) \times \mathbb{B} \rightarrow Q \times C$ generalizes to a total relation, or in other words to a function into the strict powerset of $Q \times C$, $M : (Q \times \{q_0\}) \times \mathbb{B} \rightarrow \mathcal{P}^+(Q \times C)$, that is, taking each non-halting state $q$ and current cell bit contents $b$ into a non-empty set $M(q, b)$ of non-deterministic (state,action) choices. For example, to turn the successor Turing machine in Figure 2.1 into one which non-deterministically chooses to add one more 1 to the given number or not when it reaches its end, all we have to do is to modify its transition function in state $q_1$ and cell contents 0 to return two possible continuations: $M(q_1, 0) = \{(q_2, 1), (q_2, \leftarrow)\}$. Like in Section 2.4.3, we give both lazy and unrestricted representations.

Lazy Rewrite Logic Representation

Figure 2.10 shows how a Turing machine $M$ can be associated a computationally equivalent rewrite logic theory $\mathbf{R}_{M}^{\text{lazy}}$. The only difference between this rewrite logic theory and the equational logic theory in Figure 2.2 is that the equations which were specific to the Turing machine have been turned into rewrite rules. The equation expanding the stream of zeros remains an equation. Since in rewrite logic only the rewrite rules count as transitions, and they apply modulo equations, the rewrite theory is in fact more faithful to the actual computational steps embodied in the Turing machine. The result below formalizes this by showing that there is a step-for-step equivalence between computations using $M$ and rewrites using $\mathbf{R}_{M}^{\text{lazy}}$.

Theorem 10. The rewrite logic theory $\mathbf{R}_{M}^{\text{lazy}}$ is confluent. Moreover, the Turing machine $M$ and the rewrite theory $\mathbf{R}_{M}^{\text{lazy}}$ are step-for-step equivalent, that is,

$$(q, 0^{|u| + 1} \nu v 0^{|v|}) \rightarrow_M (q', 0^{|u'| + 1} u' \nu v' 0^{|v'|}) \text{ if and only if } \mathbf{R}_{M}^{\text{lazy}} \models q(\overrightarrow{u}, b : \overrightarrow{v}) \rightarrow q'(\overleftarrow{u'}, b' : \overrightarrow{v'}) \text{.}$$
for any finite sequences of bits $u, v, u', v' \in \{0, 1\}^*$, any bits $b, b' \in \{0, 1\}$, and any states $q, q' \in Q$, where if $u = b_1b_2\ldots b_{n-1}b_n$, then $\overline{u} = b_n : b_{n-1} : \cdots : b_2 : b_1 : \text{zeros}$ and $\overline{b} = b_1 : b_2 : \cdots : b_n : \text{zeros}$. Finally, the following are equivalent:

1. The Turing machine $M$ terminates on input $b_1b_2\ldots b_n$;
2. $\mathcal{R}_M^{\text{lzy}} \models q_4(\text{zeros}, b_1 : b_2 : \cdots : b_n : \text{zeros}) \rightarrow q_4(l, r)$ for some terms $l, r$ of sort Tape; note though that $\mathcal{R}_M^{\text{lzy}}$ does not terminate on term $q_4(\text{zeros}, b_1 : b_2 : \cdots : b_n : \text{zeros})$ as an unrestricted rewrite system, since the equation $\text{zeros} = 0 : \text{zeros}$ (regarded as a rewrite rule) can apply forever, thus yielding infinite equational classes of configurations with no canonical forms, but $\mathcal{R}_M^{\text{lzy}}$ terminates on $q_4(\text{zeros}, b_1 : b_2 : \cdots : b_n : \text{zeros})$ if the stream construct operation $\langle \rangle : \langle \rangle : \text{Cell} \times \text{Tape} \rightarrow \text{Tape}$ has a lazy rewriting strategy on its second argument;

Proof. \ldots \hfill \square

Therefore, unlike the equational logic theory $\mathcal{E}_M^{\text{seq}}$ in Theorem 7, the rewrite logic theory $\mathcal{R}_M^{\text{lzy}}$ faithfully captures, step-for-step, the computational granularity of $M$. Recall that equational deduction does not count as computational, or rewrite steps in rewrite logic, which allows to apply the self-expanding equation of zeros silently in the background. Since there are no artificial rewrite steps, we can conclude that $\mathcal{R}_M$ actually is precisely $M$ and not an encoding of it. Theorem 10 thus showed not only that rewriting logic is Turing complete, but also that it faithfully captures the computational granularity of the represented Turing machines.

**Unrestricted Rewrite Logic Representations**

Figure 2.11 shows our unrestricted representation of Turing machines as rewrite logic theories, following the same idea as the equational representation in Section 2.4.3 (Figure 2.9). Let $\mathcal{R}_M$ be the rewrite logic theory in Figure 2.11 Then the following result holds:

```plaintext
\textbf{sorts:}
\textit{Cell, Tape, State, Configuration}

\textbf{operations:}
0, 1 : $\rightarrow$ Cell
zeros : $\rightarrow$ Tape
\textit{\_:\_:} : Cell $\times$ Tape $\rightarrow$ Tape
\textit{\langle\_,\_,\_\rangle} : State $\times$ Tape $\times$ Tape $\rightarrow$ Configuration
$q : \rightarrow$ State — one such constant for each $q \in Q$

\textbf{equations:}
$S(\text{zeros,} R) = S(0:\text{zeros,} R)$
$S(L, \text{zeros}) = S(L, 0:\text{zeros})$

\textbf{rules:}
$q(L, b : R) \rightarrow q'(L, b' : R)$ — one rule for each $q, q' \in Q, b, b' \in \text{Cell}$ with $(q', b') \in M(q, b)$
$q(L, b : R) \rightarrow q'(b : L, R)$ — one rule for each $q, q' \in Q, b \in \text{Cell}$ with $(q', \rightarrow) \in M(q, b)$
$q(B : L, b : R) \rightarrow q'(L, B : b : R)$ — one rule for each $q, q' \in Q, b \in \text{Cell}$ with $(q', \leftrightarrow) \in M(q, b)$

Figure 2.11: Unrestricted rewrite logic representation $\mathcal{R}_M$ of Turing machine $M$
```
Theorem 11. The rewrite logic theory $\mathcal{R}_M$ is confluent. Moreover, the Turing machine $M$ and the rewrite theory $\mathcal{R}_M$ are step-for-step equivalent, that is,

$$(q,0^\omega u b 0^\omega) \rightarrow_M (q',0^\omega u' b' v' 0^\omega) \text{ if and only if } \mathcal{R}_M \models q'(u',b':v')$$

for any finite sequences of bits $u,v,u',v' \in \{0,1\}^*$, any bits $b,b' \in \{0,1\}$, and any states $q,q' \in Q$, where if $u = b_1 b_2 \ldots b_{n-1} b_n$, then $\overrightarrow{u} = b_n : b_{n-1} : \cdots : b_2 : b_1 : \text{zeros}$ and $\overrightarrow{u} = b_1 : b_2 : \cdots : b_{n-1} : b_n : \text{zeros}$. Finally, the following are equivalent:

1. The Turing machine $M$ terminates on input $b_1 b_2 \ldots b_n$;
2. $\mathcal{R}_M$ terminates on term $q_s(\text{zeros},b_1:b_2: \cdots : b_n: \text{zeros})$ as an unrestricted rewrite system and $\mathcal{R}_M \models q_s(\text{zeros},b_1:b_2: \cdots : b_n: \text{zeros}) \rightarrow q_h(l,r)$ for some terms $l,r$ of sort Tape;

Proof. . . . □

Like for the lazy representation of Turing machines in rewriting logic discussed above, the rewrite theory $\mathcal{R}_M$ is the Turing machine $M$, in that there is a step-for-step equivalence between computational steps in $M$ and rewrite steps in $\mathcal{R}_M$. Recall, again, that equations do not count as rewrite steps, their role being to structurally rearrange the term so that rewrite rules can apply; indeed, that is precisely the intended role of the two equations in Figure 2.11 (they reveal new blank cells on the tape whenever needed). Similarly to the equational case in Section 2.4.3, the two generic equations can be completely eliminated. However, this time we have to add more Turing-machine-specific rules instead. For example, if $(q',-') \in M(q,b)$ then, in addition to the last rule in Figure 2.11 we also include the rule:

$q(\text{zeros},b:R) \rightarrow q'(\text{zeros},0:b:R)$

This way, one can expand $\text{zeros}$ and apply the transition in one rewrite step, instead of one equational step and one rewrite step. Doing that systematically for all the transitions allows us to eliminate the need for equations entirely; the price to pay is, of course, that the number of rules increases.
2.5.4 Completeness

2.5.5 Rewrite Logic as a Universal Logical Formalism

In [35], it is suggested that one generic way to incorporate any proof system in rewrite logic is to translate each deduction rule of the form

\[
\frac{Seq_1 \Seq_2 \ldots \Seq_n}{Seq}
\]

into a rewrite rule of the form

\[
Seq \rightarrow Seq_1, Seq_2, \ldots, Seq_n
\]

or into a rewrite rule of the form

\[
Seq_1, Seq_2, \ldots, Seq_n \rightarrow Seq
\]
2.5.6 Maude: A High Performance Rewrite Logic System

Maude (http://maude.cs.uiuc.edu) is a rewrite logic executable specification language, which builds upon a fast rewrite engine. Our main use of Maude in this book is as a platform to execute rewrite logic semantic definitions, following the various approaches in Chapter 3. Our goal here is to give a high-level overview of Maude, mentioning only its features needed in this book. We make no attempt to give a systematic presentation of Maude here, meant to replace its manual or other more comprehensive papers or books (some mentioned in Section 2.5.8). The features we need will be introduced on-the-fly, with enough explanations to make this book self-contained, but the reader interested in learning Maude in depth should consult its manual.

We will use Maude to specify programming language features, which, when put together via simple Maude module operations, lead to programming language semantic definitions. Since Maude is executable, interpreters for programming languages designed this way will be obtained for free, which will be very useful for understanding, refining and/or changing the languages. Moreover, formal analysis tools for the specified languages can also be obtained with little effort, such as exhaustive state-space searchers or model-checkers, simply by using the corresponding generic rewrite logic analysis tools already provided by Maude.

Devising formal semantic executable models of desired languages or tools before these are implemented is a crucial step towards a deep understanding of the language or tool. In simplistic terms, it is like devising a simulator for an expensive system before building the actual system. However, our simulators in this book will consist of exactly the semantics, or the meaning, of our desired systems, defined using a very rigorous, mathematical notation. In the obtained formal executable model of a programming language, executing a program will correspond to nothing but logical inference within the semantics of the language.

How to Execute Maude

After installing Maude on your platform and setting up the environment path variable, you should be able to type maude and immediately see a welcome screen followed by a cursor waiting for user input:

Maude>

Maude is interpreted, so you can just type your specifications and commands. However, a more practical way is to type everything in one file, say pgm.maude, and then include that file with the command

Maude> in pgm.maude

after starting Maude (the extension is optional), or, alternatively, start Maude with pgm as an argument: “maude pgm”. In both cases, the contents of pgm.maude will be loaded and executed as if it was manually typed at the cursor. Use the quit command, or simply q, to quit Maude. Since Maude’s initialization and termination are quite fast, many users end their pgm.maude file with a q command on a new line, so that Maude terminates as soon as the program is executed. Another useful command in files is eof, which tells Maude that the end-of-file is meant there and thus it does not process the code following the eof command. Instead, the control is given to the user, who can manually type commands, etc. You can correct/edit your Maude definition in pgm.maude and then load it again. However, keep it in mind that Maude maintains only one working session, in particular one module database, until you quit it. This can sometimes lead to unexpected errors for beginners, so if you are not sure about an error just quit and then restart Maude.

Modules

Maude specifications are introduced as modules. There are several kinds of modules, but for simplicity we only use general modules in this book, which have the syntax
mod <NAME> is

    <BODY>

endm

where <NAME> can be any identifier. The <BODY> of a module can include importation of other modules, sort and operation declarations, and a set of sentences. The sorts together with the operations form the signature of that module, and can be thought of as the interface of that module to other modules.

To lay the ground for introducing more Maude features, let us define Peano-style natural numbers with addition and multiplication. We define the addition first, in one separate module:

mod PEANO-NAT is

    sort Nat .
    op zero : -> Nat .
    op succ : Nat -> Nat .
    op plus : Nat Nat -> Nat .
    vars N M : Nat .
    eq plus(zero, M) = M .
    eq plus(succ(N), M) = succ(plus(N, M)) .

endm

Declarations and sentences are always terminated by periods, which should have white spaces before and after. Forgetting a terminal period or a white space before the period are two of the most common errors that Maude beginners make.

The signature of PEANO-NAT consists of one sort, Nat, and three operations, namely zero, succ, and plus. Sorts are declared with the keywords sort or sorts, and operations with op or ops.

The three operations have zero, one and two arguments, respectively, whose sorts are listed between the symbols : and ->. Operations of zero arguments are also called constants, those of one argument are called unary and those of two binary. The result sort appears right after the symbol ->.

We use ops when two or more operations of same arguments are declared together, to save space, and then we use white spaces to separate them:

    ops plus mult : Nat Nat -> Nat .

There are few special characters in Maude, and users are allowed to define almost any token or combination of tokens as operation names. If you use op in the above instead of ops, for example, then only one operation, called “plus mult”, is declared.

The two equations in PEANO-NAT are properties, or constraints, that terms built with these operations must satisfy. Another way to look at equations is through the lenses of possible implementations of the specifications they define; in our case, any correct implementation of Peano natural numbers should satisfy the two equations. Equations are quantified universally with the variables they contain, and can be applied from left-to-right or from right-to-left in reasoning, which means that equational proofs may require exponential search, thus making them theoretically intractable. Maude provides limited support for equational reasoning.

reduce: Rewriting with Equations

When executing specifications, Maude regards all equations as rewrite rules, which means that they are applied only from left to right. Moreover, they are applied iteratively for as long as their left-hand-side terms match any subterm of the term to reduce. This way, any well-formed term can either be derived infinitely often, or be reduced to a normal form, which cannot be reduced anymore by applying equations as rewriting rules. Maude’s command to reduce a term to its normal form using equations as rewrite rules is reduce, or simply red. Reduction will be made in the last defined module, which is PEANO-NAT in our case:
Maude> reduce plus(plus(succ(zero), succ(succ(zero))), succ(succ(succ(zero)))) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: succ(succ(succ(succ(succ(succ(zero)))))

Make sure commands are terminated with a period. Maude implements state of the art term rewriting algorithms, based on advanced indexing and pattern matching techniques. This way millions of rewrites per second can be performed, making Maude usable as a programming language in terms of performance.

Sometimes the results of reductions are repetitive and may be too large to read. To ameliorate this problem, Maude provides an operator attribute called \texttt{iter}, which allows to input and print repetitive terms more compactly. For example, if we replace the declaration of operation \texttt{succ} with

\begin{verbatim}
  op succ : Nat -> Nat [iter].
\end{verbatim}

then Maude uses, e.g., \texttt{succ^3(zero)} as a shorthand for \texttt{succ(succ(succ(zero)))}. For example,

Maude> reduce plus(plus(succ(zero), succ^2(zero)), succ^3(zero)) .
result Nat: succ^6(zero)

\section*{Importation}

Modules can be imported in several different ways. The difference between importation modes is subtle and semantical rather than operational, and it is not relevant in this book. Therefore, we only use the most general of them, \texttt{including}. For example, the following module extends \texttt{PEANO-NAT} with multiplication:

\begin{verbatim}
mod PEANO-NAT* is
  including PEANO-NAT .
  op mult : Nat Nat -> Nat .
  vars M N : Nat .
  eq mult(zero, M) = zero .
  eq mult(succ(N), M) = plus(mult(N, M), M) .
endm
\end{verbatim}

It is safe to think of \texttt{including} as “copy and paste” the contents of the imported module into the importing module, with one exception: variable declarations are \textit{not} imported, so they need to be redeclared.

We can now “execute programs” using features in both modules:

\begin{verbatim}
red mult(plus(succ(zero), succ(succ(zero))), succ(succ(succ(zero)))) .
\end{verbatim}

The following is Maude’s output:

rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: succ^9(zero)

Even though this language is very simple and its syntax is ugly, it nevertheless shows a formal and executable definition of a language using equational logic and rewriting. Other languages or formal analyzers discussed in this book will be defined in a relatively similar manner, though, as expected, they will be more involved.

\section*{The Mixfix Notation and Parsing}

The \texttt{plus} and \texttt{mult} operations defined above are meant to be written using the \textit{prefix} notation in terms. Maude also supports the \textit{mixfix} notation for operations (see Section \ref{s:mixfix}), by allowing the user to write underscores in operation names as placeholders for their corresponding arguments.
Users can now also write terms taking advantage of the mixfix notation, for example 3 + 5, in addition to the usual prefix notation, that is, \(+\)(3, 5).

Recall from Section 2.1.3 that, syntactically speaking, the mixfix notation has the same expressiveness as the context-free grammar notation. Therefore, the mixfix notation comes with the unavoidable parsing problem. For example, suppose that we replace the operations \(+\) and \(*\) in the modules above with their mixfix variants \(+\) and \(*\) (see also Exercise 29). Then the term \(X + Y * Z\), with \(X, Y, Z\) arbitrary variables (or any terms) of sort \(Nat\), admits two ambiguous parsings: \((X + Y) * Z\) and \(X + (Y * Z)\).

Maude provides a `parse` command, similar to `reduce` except that it only parses the given term:

```
Maude> parse X + Y .
Nat: X + Y
```

Maude generates a warning message whenever it detects more than one parsing of the given term:

```
Maude> parse X + Y * Z .
Warning: <standard input>, line 1: ambiguous term, two parses are:
X + (Y * Z)
-versus-
(X + Y) * Z
Arbitrarily taking the first as correct.
Nat: X + (Y * Z)
```

Similar warning messages are issued when ambiguous terms are detected in the specification (e.g., in equations). In general, we do not want to allow any parsing ambiguity in specifications or in terms to rewrite. One simple way to avoid ambiguities is to use parentheses to specify the desired grouping, for example:

```
Maude> parse X + (Y * Z) .
Nat: X + (Y * Z)
```

To reduce the number of parentheses, Maude allows us to assign precedences to mixfix operations declared in its modules, specifically as operator attributes in square brackets using the `prec` keyword. For example:

```
```

*The lower the precedence the stronger the binding!* As expected, now there is no parsing ambiguity anymore:

```
Maude> parse X + Y * Z .
Nat: X + Y * Z
```

To see how the term was parsed, set the “print with parentheses” flag on:

```
Maude> set print with parentheses on .
Maude> parse X + Y * Z .
Nat: (X + (Y * Z))
```

If displaying the parentheses is not sufficient, then disable the mixfix printing completely:

```
Maude> set print mixfix off .
Maude> parse X + Y * Z .
Nat: _+_(X, _*_(Y, Z))
```
Associativity, Commutativity and Identity Attributes

Some of the binary operations used in this book will be associative (A), commutative (C) or have an identity (I), or combinations of these. E.g., \( + \) is associative, commutative and has 0 as identity. All these can be added as attributes to operations when declared:

\[
\begin{align*}
\text{op } \_+\_ & : \text{Int Int} \rightarrow \text{Int} \ [\text{assoc comm id: 0 prec 33}] . \\
\text{op } \_\times\_ & : \text{Int Int} \rightarrow \text{Int} \ [\text{assoc comm id: 1 prec 31}] .
\end{align*}
\]

Note that each of the A, C, and I attributes are logically equivalent to appropriate equations, such as

\[
\begin{align*}
eq A + (B + C) &= (A + B) + C . \\
eq A + B &= B + A . & \quad \text{---&gt; attention: rewriting does not terminate!} \\
eq A + 0 &= A .
\end{align*}
\]

When applied as rewrite rules, each of the three equations above have limitations. The associativity equation forces all the parentheses to be grouped to the left, which may prevent some other rules from applying. The commutativity equation may lead to non-termination when applied as a rewrite rule. The identity equation would only be able to simplify expressions, but not to add a 0 to an expression, which may be useful in some situations (we will see such situations shortly, in the context of lists). Maude’s builtin support for ACI attributes addresses all the problems above. Additionally, the \text{assoc} attribute of a mixfix operation is also taken into account by Maude’s parser, which hereby eliminates the need for some useless parentheses:

\[
\begin{align*}
\text{Maude} & \text{> parse } X + Y + Z . \\
\text{Nat} & : X + Y + Z
\end{align*}
\]

An immediate consequence of the builtin support for the \text{comm} attribute, which allows rewriting with commutative operations to terminate, is that normal forms will be reported now \text{modulo commutativity}:

\[
\begin{align*}
\text{Maude} & \text{> red } X + Y + X . \\
\text{rewrites} : 0 & \text{ in } 0\text{ms cpu (0ms real) } (\sim \text{ rewrites/second}) \\
\text{result Nat} & : X + X + Y
\end{align*}
\]

As seen, Maude picked to display some equivalent (modulo AC) of the original term (extracted from how the current implementation of Maude stores this term internally). There were 0 rewrites applied in the reduction above, because the internal rearrangements of terms according to the ACI attribute annotations do not count as rule applications.

Matching Modulo Associativity, Commutativity, and Identity

Here we discuss Maude’s support for ACI matching, which is arguably one of the most distinguished and complex Maude features, and nevertheless the reason and the most important use of the ACI attributes.

We discuss ACI matching by means of a series of examples, starting with lists, which occur in many programming languages. The following module defines lists of integers with a membership operation \( \_\in\_ \), based on AI (associative and identity) matching:

\[
\begin{align*}
\text{mod INT-LIST is including INT} . \\
\text{sort IntList} . \\
\text{subsort Int} < \text{IntList} . \\
\text{op nil : } \rightarrow \text{IntList} . \\
\text{op \_\_ : IntList IntList} \rightarrow \text{IntList} [\text{assoc id: nil}] . \\
\text{op \_\_\_ : Int IntList} \rightarrow \text{Bool} . \\
\text{var I : Int} . \ \text{vars L L' : IntList} . \\
\text{eq I in L I L' = true} . \\
\text{eq I in L = false [owise]} .
\end{align*}
\]

endm
We start by including the builtin INT module, which declares a sort Int and provides arbitrary large integers as constants of sort Int, together with the usual operations on these. The builtin module BOOL, which similarly declares a sort Bool and common Boolean operations on it, is automatically included in all modules, so it needs not be included explicitly. To see an existing module, builtin or not, use the command

Maude> show module <NAME> .

For example, “show module INT .” will display the INT module. In the INT-LIST module above, note the subsort declaration “Int < IntList”, which says that integers are also lists of integers. This, together with the constant nil and the concatenation operation _, can generate any finite list of integers:

Maude> parse 1 2 3 4 5 .
   IntList: 1 2 3 4 5
Maude> red 1 nil 2 nil 3 nil 4 nil 5 6 7 nil .
   rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
   result IntList: 1 2 3 4 5 6 7

Note how the reduce command above eliminated all the unnecessary nil constants from the list, in zero rewrite steps, for the same reason as above: the internal rearrangements according to the ACI attributes do not count as rewrite steps.

The two equations defining the membership operation make use of AI matching. The first equation says that if we can match the integer I anywhere inside the list, then we are done. Since the list constructor _ was declared associative and with identity nil, Maude is mathematically allowed to bind the variables L and L’ of sort IntList to any lists of integers, including the empty one. Maude indeed does this through its efficient AI matching algorithm. Equations with attribute owise are applied only when other equations fail to apply. Therefore, we defined the semantics of the membership operation only by means of AI matching, without having to implement any explicit traversal of the list. Here are some examples testing the semantics above:

Maude> red 3 in 2 3 4 .
   result Bool: true
Maude> red 3 in 3 4 5 .
   result Bool: true
Maude> red 3 in 1 2 4 .
   result Bool: false

To define sets of integers (see, e.g., Exercise 30), besides likely renaming the sort IntList into IntSet, we would also need to declare the concatenation operation _ commutative; moreover, thanks to Maude’s commutative matching, we can also replace the first equation by “eq I in I L = true .”

We next discuss a Maude definition of (partial finite-domain) maps (see Section 2.4.6 and Figure 2.7 for the mathematical definition). We assume that the Source and Target sorts are defined in separate modules SOURCE and TARGET, respectively; one may need to change these in concrete applications. The associativity, commutativity and identity equations in Figure 2.7 are replaced by corresponding Maude operational attributes. Note that the second equation defining the update operation takes advantage of Maude’s owise attribute (explained above), so it departs from the more mathematical definition in Figure 2.7.

mod MAP is including SOURCE + TARGET .
   sort Map .
   op _|->_ : Source Target -> Map [prec 0] .
   op empty : -> Map .
   op _-_ : Map Map -> Map [assoc comm id: empty] .
   op _(__) : Map Source -> Target [prec 0] . --- lookup
   op _[/_/] : Map Target Source -> Map [prec 0] . --- update
var M : Map . var A : Source . var B B' : Target .
eq (M, A |-> B)(A) = B .
eq (M, A |-> B')[B / A ] = (M, A |-> B) .
eq M[B / A ] = (M, A |-> B) [owise] .
endm

If module SOURCE defines constants a, b, c, d, ..., of sort Source, and TARGET defines constants 1, 2, 3, 4, ..., of sort Target, then the following reduce commands work as shown:

Maude> red empty[1 / a][2 / b][3 / c] .
result Map: a |-> 1, b |-> 2, c |-> 3
Maude> red empty[1 / a][2 / b][3 / c][4 / a] .
result Map: a |-> 4, b |-> 2, c |-> 3
Maude> red empty[1 / a][2 / b][3 / c][4 / a](a) .
result Target: 4
Maude> red empty[1 / a][2 / b][3 / c][4 / a](d) .
result Target: (a |-> 4, b |-> 2, c |-> 3)(d)

Note that the last reduction above only updated the map, but it got stuck on the lookup of d. That is because we only have one equation defining lookup, which works only when the looked up element is in the domain of the map. Getting stuck terms as above may be acceptable in many applications, but, however, we sometimes want to report specific errors in such situations. Maude has several advanced foundational mechanisms to deal with errors, but they are non-trivial and we do not need them in this book. Instead, we can simply modify the MAP definition above to include a special undefined “value” in the Target domain and then explicitly use this value in equations where we mean that an error occurred:

op undefined : -> Target .
eq M(A) = undefined [owise] .

Now the last reduction command above yields undefined. A particularly useful approach to deal with undefinedness in the context of programming language semantics, where a semantics built upon several mathematical domains in which the syntax is interpreted, is to define the constant undefined to have a generic sort, say Undefined, which is then subsorted to all sorts, including Source, Target, and Map. Then we can add the following to the module MAP to obtain partial finite-domain maps with support for undefinedness:

subsort Undefined < Map .
eq M(A) = undefined [owise] .
eq A |-> undefined = empty .

The last equation above is an optimization, allowing us to “garbage collect” the useless bindings in maps once they are explicitly “undefined” in certain elements. For example,

Maude> red empty[1 / a][2 / b][3 / c][4 / a][undefined / a] .
result Map: b |-> 2, c |-> 3

Pretty Printing

In the MAP example above, the bindings and the comma separating them may be hard to read when the maps are large. We may therefore want to pretty-print the reduced terms. Maude provides the format attribute for this purpose. For example, if we replace the operation declarations of _|->_ and _,_, with

op _|->_ : Source Target -> Map [prec 0 format(d b o d)] .
op _,_ : Map Map -> Map [assoc comm id: empty format(d sr! oss d)] .

68
then the former will always be displayed in color blue, while the second in bold red and preceded by one space and followed by two spaces. Each group of characters in the argument of format refers to a pointcut in the operation name; we have default pointcuts at the beginning and at the end of the operation name, as well pointcuts before and after any special token (underscore, comma, and the various kinds of parentheses and brackets). In each group of characters, \( d \) means “default” and is used alone to skip that pointcut and move to the next, \( b \) and \( r \) the colors blue and red, \( o \) means to revert to the original color and style, \( s \) means one space, and \( ! \) means bold font. There are also indentation attributes, which we have not used here but we will use later in the book, such as: + and − to increment and decrement the global indent counter, respectively, \( i \) to print the number of spaces determined by indent counter, and \( n \) to print a newline.

**Built-in Modules**

Maude provides several builtin modules and has been designed in a way that existing modules can be easily changed and more modules can be added. It is likely that at this moment Maude’s builtin modules are not identical to the homonymous ones when this book was written, and it also likely that new modules have been added since then. To avoid depending on particular versions of Maude, and also to avoid unfortunate naming conflicts with existing builtins which prevent us from naming programming language constructs as desired, in this book we actually define a custom version builtins, discussed in Section [A.1]. Nevertheless, some of Maude’s current builtin modules make interesting use of ACI matching and are also present in our custom builtins, albeit using different names, so we briefly discuss them here.

As already discussed, the INT module provides a sort \( \text{Int} \) with arbitrarily large integers and common operations on them. All these are essentially hooked to C library functions through a special interface that Maude provides, which we do not discuss here but refer the interested reader to Section [A.1] for details.

Similarly, there is a builtin module named QID, from “quoted identifiers”, which provides a sort \( \text{Qid} \) together with arbitrary large quoted identifiers, as constants of sort \( \text{Qid} \), such as the following: ‘a’, ‘b’, ‘abc123’, ‘a-larger-identifier’, etc. These can be used as identifiers, e.g., as program variable names, in the programming languages that we define and execute using Maude.

Let us next discuss the module BOOL, which by default Maude includes in every other module (there are ways to disable the automatic inclusion, discussed in Section [A.1]). Besides the sort \( \text{Bool} \) with its two Boolean constants \( \text{true} \) and \( \text{false} \), BOOL includes three important *polymorphic* operations and the usual Boolean operations. The polymorphic operations have the following syntax:

\[
\text{op if\_then\_else\_fi} : \text{Bool Universal Universal} \rightarrow \text{Universal} [...] \\
\text{op \_==\_} : \text{Universal Universal} \rightarrow \text{Bool} [...] \\
\text{op \_=/=_} : \text{Universal Universal} \rightarrow \text{Bool} [...] 
\]

We excluded their specific attributes because they use advanced Maude features which are not necessary anywhere else in this book. Instead, we explain their behavior in words. The builtin sort \( \text{Universal} \) can be thought of as a generic sort, which can be instantiated with any concrete sort. Operation \( \text{if\_then\_else\_fi} \) is strict in its first argument and lazy in its second and third arguments, that is, it only allows rewrites to take place in its first argument but not in the other two arguments. If the first argument rewrites to \( \text{true} \) then the conditional rewrites to its second argument, and if the first argument rewrites to \( \text{false} \) then the conditional rewrites to its third argument. We will discuss rewrite strategies in more depth later in this section. The other two operations correspond to operational equality and inequality of terms: the two terms are first rewritten to their normal forms and then those are compared modulo the existing operation ACI attributes. While operational equality implies logical equality, the other implication does not hold and tends to be a source of confusion, sometimes even among Maude experts: two terms \( t \) and \( t' \) may be provably equal using operational
reasoning, yet \( t = t' \) may still rewrite to false. In this case, false only means that Maude was not able to show the two terms equal by rewriting them to their normal forms.

The definitions of the usual Boolean operations make interesting use of AC matching:

\[
\begin{align*}
\text{op} & \ _\text{and}_\ : \ 	ext{Bool} \times \text{Bool} \rightarrow \text{Bool} \ [\text{assoc} \ 	ext{comm} \ 	ext{prec} \ 55] . \\
\text{op} & \ _\text{or}_\ : \ 	ext{Bool} \times \text{Bool} \rightarrow \text{Bool} \ [\text{assoc} \ 	ext{comm} \ 	ext{prec} \ 59] . \\
\text{op} & \ _\text{xor}_\ : \ 	ext{Bool} \times \text{Bool} \rightarrow \text{Bool} \ [\text{assoc} \ 	ext{comm} \ 	ext{prec} \ 57] . \\
\text{op} & \ _\text{not}_\ : \ 	ext{Bool} \rightarrow \text{Bool} \ [\text{prec} \ 53] . \\
\text{op} & \ _\text{implies}_\ : \ 	ext{Bool} \times \text{Bool} \rightarrow \text{Bool} \ [\text{prec} \ 61 \ 	ext{gather} (e \ E)] .
\end{align*}
\]

vars A B C : Bool .

eq \text{true} \ _\text{and}_\ A = A .

eq \text{false} \ _\text{and}_\ A = \text{false} .

eq \text{A and A} = \text{A} .

eq \text{false xor A} = \text{A} .

eq \text{A xor A} = \text{false} .

eq \text{A and (B xor C)} = \text{A and B xor A and C} .

eq \text{not A} = \text{A xor true} .

eq \text{A or B} = \text{A and B xor A xor B} .

eq \text{A implies B} = \text{not (A xor A and B)} .
\]

It can be shown that the equations above, when applied as rewrite rules, yield a decision procedure for propositional logic. Specifically, if we only consider Bool terms which are variables, any valid proposition rewrites to true and any unsatisfiable one to false; the remaining (satisfiable but invalid) propositions are rewritten to a canonical form consisting of an exclusive disjunction (xor) of conjunctions (and). We refer the interested reader to Section 2.3 for terminology and basic propositional logic results.

The attribute gather(e E) of _implies_ tells the Maude parser that we want _implies_ to be right associative, this way avoiding to add unnecessary parentheses in Boolean expressions. There is a similar attribute, gather(E e), for left associativity. We do not discuss the gather attribute any further in this book.

**Constructor versus Defined Operations**

Recall the two equations of the module PEANO-NAT:

\[
\begin{align*}
\text{eq} & \ \text{plus} (\text{zero}, M) = M . \\
\text{eq} & \ \text{plus} (\text{succ} (N), M) = \text{succ} (\text{plus} (N, M)) .
\end{align*}
\]

These equations constrain the three operations of the module, namely

\[
\begin{align*}
\text{op} & \ \text{zero} : \rightarrow \text{Nat} . \\
\text{op} & \ \text{succ} : \text{Nat} \rightarrow \text{Nat} . \\
\text{op} & \ \text{plus} : \text{Nat Nat} \rightarrow \text{Nat} .
\end{align*}
\]

But why did we write them in that particular style, which resembles a recursive definition of a function plus in terms of data-type constructors zero and succ? Intuitively, that is because we want plus to be completely defined in terms of zero and succ. Formally, this means that any term over the syntax zero, succ, and plus can be shown, using the given equations, equal to a term containing only zero and succ operations, that is, zero and succ alone are sufficient to build any Peano number.

While Maude at its core makes no distinction between operations meant to be data-type constructors and operations meant to be functions, it is still meaningful to distinguish the operations of a module into constructor and defined operations. Note, however, that it is the way we write the equations in the module, and only that, which makes the operations become constructors or defined. If we forget an equation dealing with a case (e.g., an intended constructor) for an operation intended to be defined, then that operation cannot
be always eliminated from terms, so technically speaking it is also a constructor. Unfortunately, there is no silver-bullet recipe on how to define “defined” operators, but essentially a good methodology is to define the operator’s behavior on each intended constructor. That is what we did when we defined plus in PEANO-NAT and mult in PEANO-NAT*: we defined them on zero and on succ. In general, if c1, . . . , cn are the intended constructors of a data-type, in order to define a new operation d, make sure that all equations of the form

\[
\begin{align*}
\text{eq } d(c_1(...)) &= \ldots \\
\text{...} \\
\text{eq } d(c_n(...)) &= \ldots
\end{align*}
\]

are in the specification. If d has more arguments, then make sure that the above cases are listed for at least one of its arguments. This gives no guarantee (e.g., one can “define” plus as \(\text{plus}(\text{succ}(M),N) = \text{plus}(\text{succ}(M),N)\)), but it is a good enough principle to follow.

Let us demonstrate the above by defining several operations. Consider the following specification of lists:

\[
\begin{align*}
\text{mod } \text{INT-LIST} &\text{ is including } \text{INT} . \\
\text{sort } \text{IntList} . \text{ subsort } \text{Int} < \text{IntList} . \\
\text{op } \_ : \text{Int} \times \text{IntList} \rightarrow \text{IntList} [\text{id: nil}] . \\
\text{op } \text{nil} : \rightarrow \text{IntList} . \\
\text{endm}
\end{align*}
\]

The two operations are meant to be constructors for lists, namely the empty list and adding an integer to the beginning of a list. Note, however, that the above contains three constructors for lists at first sight, because the subsorting of Int to IntList states that sole integers are also lists. Indeed, without the identity attribute of _, we would have to consider three cases when defining operations over lists. However, with the identity declaration Maude will internally identity integers I with lists I nil, so only two constructors are needed, corresponding to the two declared operations. Let us next define several important and useful operations on lists. Notice that the definition of each operator treats each of the two constructors separately.

The following defines the usual list length operator:

\[
\begin{align*}
\text{mod } \text{LENGTH} &\text{ is including } \text{INT-LIST} . \\
\text{op } \text{length} : \text{IntList} \rightarrow \text{Nat} . \\
\text{var } I : \text{Int} . \text{ var } L : \text{IntList} . \\
\text{eq } \text{length}(I L) = 1 + \text{length}(L) . \\
\text{eq } \text{length}(\text{nil}) = 0 . \\
\text{endm}
\end{align*}
\]

\text{red } \text{length}(1 \ 2 \ 3 \ 4 \ 5) . \quad ***> \text{ should be 5}

The following defines list membership, without speculating matching (in fact, this would not be possible anyway because concatenation is not defined associative as before):

\[
\begin{align*}
\text{mod } \text{IN} &\text{ is including } \text{INT-LIST} . \\
\text{op } \_\text{in}_\_ : \text{Int} \times \text{IntList} \rightarrow \text{Bool} . \\
\text{vars } I \ J : \text{Int} . \text{ var } L : \text{IntList} . \\
\text{eq } I \text{ in } J L = \text{if } I == J \text{ then true else } I \text{ in } L \text{ fi} . \\
\text{eq } I \text{ in } \text{nil} = \text{false} . \\
\text{endm}
\end{align*}
\]

\text{red } 3 \text{ in } 2 \ 3 \ 4 . \quad ***> \text{ should be true}
\text{red } 3 \text{ in } 3 \ 4 \ 5 . \quad ***> \text{ should be true}
\text{red } 3 \text{ in } 1 \ 2 \ 3 . \quad ***> \text{ should be true}
\text{red } 3 \text{ in } 1 \ 2 \ 4 . \quad ***> \text{ should be false}

71
The next defines list append:

```plaintext
mod APPEND is including INT-LIST.
  op append : IntList IntList -> IntList.
  var I : Int. vars L1 L2 : IntList.
  eq append(I L1, L2) = I append(L1, L2).
  eq append(nil, L2) = L2.
endm

red append(1 2 3 4, 5 6 7 8). *** should be 1 2 3 4 5 6 7 8
```

Notice that append has two arguments and that we have picked the first one to define our cases on. One can still show that append is a defined operation, in that it can be eliminated by equational reasoning from any term of sort IntList. The following imports APPEND and defines an operation which reverses a list:

```plaintext
mod REV is including APPEND.
  op rev : IntList -> IntList.
  var I : Int. var L : IntList.
  eq rev(I L) = append(rev(L), I).
  eq rev(nil) = nil.
endm

red rev(1 2 3 4 5). *** should be 5 4 3 2 1
```

The next module defines an operation, isort, which sorts a list of integers by insertion sort:

```plaintext
mod ISORT is including INT-LIST.
  op isort : IntList -> IntList.
  vars I J : Int. var L : IntList.
  eq isort(I L) = insert(I, isort(L)).
  eq isort(nil) = nil.
  op insert : Int IntList -> IntList.
  eq insert(I, J L) = if I > J then J insert(I, L) else I J L fi.
  eq insert(I, nil) = I.
endm

red isort(4 7 8 1 4 6 9 2 8 3 2 7 9). *** should be 1 2 2 3 4 4 6 7 7 8 8 9 9
```

An auxiliary insert operation is also defined, which takes an integer and a sorted list and rewrites to a sorted list inserting the integer argument at its place in the list argument. Notice that this latter operation makes use of the builtin if_then_else_fi operation provided by the default BOOL module discussed above, as well as of the integer comparison operation “>” provided by the builtin module INT.

Let us now consider binary trees, where a tree is either empty or an integer with a left and a right subtree:

```plaintext
mod TREE is including INT.
  sort Tree.
  op ___ : Tree Int Tree -> Tree.
  op empty : -> Tree.
endm
```

We next define some operations on trees, following the tree structure given by the two constructors above. The next operation mirrors a tree, i.e., it replaces left subtrees by the mirrored right siblings and vice-versa:

```plaintext
mod MIRROR is including TREE.
  op mirror : Tree -> Tree.
  vars L R : Tree. var I : Int.
```
eq mirror(L I R) = mirror(R) I mirror(L) .
eq mirror(empty) = empty .
endm

red mirror((empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
*** should be (empty 2 (empty 6 empty)) 5 ((empty 1 empty) 3 empty)

Searching in binary trees can be defined as follows:

mod SEARCH is including TREE .
op search : Int Tree -> Bool .
vars I J : Int . vars L R : Tree .
eq search(I, L I R) = true .
ceq search(I, L J R) = search(I, L) or search(I, R) if I /= J .
eq search(I, empty) = false .
endm

red search(6, (empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) . ***> should be true
red search(7, (empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) . ***> should be false

Note that we used a conditional equation above. Conditional equations are introduced with the keyword ceq, and their condition with the keyword if. There are several types of conditions in Maude, which we will discuss in the sequel, as needed. Here we used the simplest of them, namely a Bool term. To be faithful to rewrite logic (Section 2.5), we can regard a Boolean condition \( b \) as syntactic sugar for the equality \( b = \text{true} \); in fact, Maude also allows us to write \( b = \text{true} \) instead of \( b \). We can combine the first two equations above into an unconditional one, using an if then else fi in its RHS (see Exercise 31).

We next define a module which imports both modules of trees and of lists on integers, and defines an operation which takes a tree and returns the list of all integers in that tree, in an infix traversal:

mod FLATTEN is
  including APPEND .
  including TREE .
op flatten : Tree -> IntList .
vars L R : Tree . var I : Int .
eq flatten(L I R) = append(flatten(L), I flatten(R)) .
eq flatten(empty) = nil .
endm

red flatten((empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) . ***> should be 3 1 5 6 2

Reduction Strategies

We sometimes want to inhibit the application of equations on some subterms, for executability reasons. For example, imagine a conditional expression construct if_then_else_ in a pure language or calculus, i.e., one whose expression evaluation has no side-effects, say \( \lambda \)-calculus (this is discussed extensively in Section 4.5), whose semantics is governed by equations. While it is semantically safe to apply equations anywhere at any time, including inside the two branches of the conditional, for executability reasons we may prefer not to. Indeed, e.g., any reduction step applied in the negative branch would be a waste of computation if the condition turns out to evaluate to true. Worse, the wasteful reduction steps may lead to computational non-termination and resource exhaustion without giving the conditional a chance to evaluate its condition and then discard the non-terminating branch.

Because of reasons like above, many executable equational logic systems provide support for reduction strategies. In Maude, reduction strategies are given as operator strat attributes taking as argument sequences
of numbers. For example, the conditional operation would be assigned the strategy \texttt{strat(1 0)}, meaning that the first argument of the conditional is reduced to its normal form (the $1$), and \textit{then} the conditional itself is allowed to be reduced (the $0$); since 2 and 3 are not mentioned, the second and the third argument of the conditional are never reduced while the conditional statement is still there. By default, operations of \textit{n} arguments have strategy \texttt{strat(1 2...n 0)}, which yields a leftmost innermost reduction strategy.

Let us next discuss an interesting application of rewrite strategies in the context and lazy, infinite data-structures. One of the simplest such structure is the \textit{stream}, that is, the infinite sequence. Specifically, we consider streams of integers, as defined in Figure \ref{fig:stream}. The key idea is to assign the stream construct $\texttt{::}$, which adds a given integer to the beginning of a given stream, the reduction strategy \texttt{strat(1 0)}. In other words, reduction is inhibited in the tail of streams built using the $\texttt{::}$ This allows us to have finite representations of infinite streams. To “observe” the actual elements of such stream structures, we define an operation $\#$ taking a natural number $N$ and a stream $S$, and unrolling $S$ until its first $N$ elements become available. The module below defines this basic stream infrastructure, as well as some common streams and operations on them:

```maude
mod STREAM is including INT .
  sort Stream .
  op $\texttt{::}$ : Int Stream -> Stream [strat(1 0)] .
  var N : Int . var S S' : Stream .
  op h : Stream -> Int . eq h(N : S) = N .
  op t : Stream -> Stream . eq t(N : S) = S .
  op # : Nat Stream -> Stream . ***> #(N, S) displays the first N elements of stream S
  eq #(1, S) = h(S) : t(S) .
  ceq #(s(N), S) = h(S) : S' if S' := #(N, t(S)) .
  op zeros : -> Stream . eq zeros = 0 : zeros .
  op ones : -> Stream . eq ones = 1 : ones .
  op nat : Nat -> Stream . eq nat(N) = N : nat(N + 1) .
  op zip : Stream Stream -> Stream . eq zip(S, S') = h(S) : zip(S',t(S)) .
  op blink : -> Stream . eq blink = zip(zeros,ones) .
  op add : Stream -> Stream . eq add(S) = (h(S) + h(t(S))) : add(t(t(S))) .
  op fibonacci : -> Stream . eq fibonacci = 0 : 1 : add(zip(fibonacci,t(fibonacci))) .
endm
```

The definition of the basic observers $h$ (head) and $t$ (tail) is self-explanatory. The definition of the general observer $\#$ involves a conditional equation, and that equation has a \textit{matching} ($\texttt{=}\) in its condition. In terms of equational logic, the matching symbol $\texttt{=}\) is nothing but the equality $\texttt{=}\); however, because of its special operational nature allowing us to introduce variables that do not appear in the LHS of the equation, Maude prefers to use a different symbol for it. The streams \texttt{zeros} and \texttt{ones}, of infinitely many 0 and 1 sequences, respectively, can be defined so compactly and then executed exactly because of the reduction strategy of $\texttt{::}$, which disallows the uncontrolled, non-terminating unrolling of these streams. However, we can observe as many elements of these streams as we wish, say 7, using controlled unrolling as follows:

```maude
red #(7, zeros) . ***> 0 : 0 : 0 : 0 : 0 : 0 : t(0 : zeros)
red #(7, ones) . ***> 1 : 1 : 1 : 1 : 1 : 1 : t(1 : ones)
```

Note how the reduction strategy enables the tails of the streams above, starting with their 8-th element, to stay unreduced, thus preventing non-termination.

The module above also defines the stream of natural numbers, an operation \texttt{zip} which interleaves two streams element by element, and operation \texttt{add} which generates a new stream by adding any two consecutive elements of a given stream, and finally two concrete streams defined using \texttt{zip}: one called \texttt{blink} which zips the streams \texttt{zeros} and \texttt{ones}, and one called \texttt{fibonacci} which contains the Fibonacci sequence (see also Exercise \ref{exercise:25}). Here are some sample reductions with these streams:
To save space, we did not show the remaining tail terms for the last three reductions.

**Rewrite Rules**

Until now we have only discussed one type of Maude sentences, its equations, and how to perform reductions with them regarded as rewrite rules, oriented from left to right. As seen in Section 2.5.1, rewrite logic has two types of sentences: equations and rewrite rules. Semantically, the rewrite rules establish transitions between equivalence classes of terms obtained using the equations. In other words, the distinction between equations and rewrite rules is that the former can be applied in any direction and do not count as computational steps in the transition systems associated to terms, while the latter can only be irreversibly applied from left to right and count as transitions. Because of the complexity explosion resulting from applying equations bidirectionally, Maude applies them in very restricted but efficient ways, which we have already discussed: equations like associativity, commutativity and identity are given by means of operator attributes and are incorporated within Maude’s internal ACI rewrite algorithm, while other equations are only applied from left to right, same like the rewrite rules. While there is no visible distinction in Maude between equations and rewrite rules in terms of executability, several of Maude’s tools, including the `search` command discussed below, treat them differently, so it is important to understand the difference between them.

The following module, inspired from the Maude manual, models a simple vending machine selling coffee for one dollar and tea for three quarters. The machine takes only one-dollar coins as input ($). For one dollar one can either buy one coffee, or one tea with a quarter rest. Assume that an external agent can silently and automatically change money as needed, that is, four quarters into a dollar (the viceversa is useless here); in other words, changing money does not count as a transaction in this simplistic vending machine model.

```maude
mod VENDING-MACHINE is
  sorts Coin Item State .
  subsorts Coin Item < State .
  op null : -> State .
  op $ : -> Coin [format (r! o)] .
  op q : -> Coin [format (r! o)] .
  op tea : -> Item [format (b! o)] .
  op coffee : -> Item [format (b! o)] .

  rl $ => coffee .
  rl $ => tea q .
  eq q q q q = $ .
endm
```

All the coins and all the items that one has at any given moment forms a `State`. The two rules above model the two actions that the machine can perform, and the equation models the money changing agent.

A first distinction between equations and rules can be seen when using the `reduce` command:

```
Maude> red $ q q q .
result State: $ q q q
```

Note that the equation cannot reduce this state any further when applied from left to right, and that no rewrite rules were applied. Indeed, the command `reduce` only applies equations, and only from left-to-right. To apply both equations *and* rewrite rules, we have to use the command `rewrite` or its shortcut `rew`:
Maude> rew $ q q q .
result State: q q q coffee

Maude chose to apply the first rule and the resulting state cannot be rewritten or reduced anymore, so we are stuck with three useless quarters. If Maude had chosen the second rule, then we could have bought both a coffee and a tea with our money. To see all possible ways to rewrite a given term, we should use the search command instead of rewrite:

Maude> search $ q q q =>! S:State .
Solution 1 (state 1)
S:State --> q q q coffee
Solution 2 (state 3)
S:State --> tea coffee
Solution 3 (state 4)
S:State --> q tea tea

The search command takes a term and a pattern and attempts to systematically (in breadth-first order) apply the rewrite rules on the original term in order to match the pattern. In our case, the pattern is a state variable $S$, so all states match it. The term and the pattern are separated by a decorated arrow. Different decorations mean different things. The $!$ above means that we are interested only in normal forms. Indeed, the above search command has precisely three solutions, as reported by Maude. Another useful decoration is $*$, which shows all intermediate states, not only the normal forms:

Maude> search $ q q q =>* S:State .
Solution 1 (state 0)
S:State --> $ q q q
Solution 2 (state 1)
S:State --> q q q coffee
Solution 3 (state 2)
S:State --> $ tea
Solution 4 (state 3)
S:State --> tea coffee
Solution 5 (state 4)
S:State --> q tea tea

Notice that we never see four quarters in a state in the solutions above, in spite of the $*$ decoration; the equation automatically changed them into a dollar. Remember: equations do not count as transitions, so their application is not visible in the transition system explored by search. Another way to think of it is rewrite rules apply modulo equations; that is, equations structurally rearrange the term so that rules match and apply. Yet another way to think about it is that equations take time zero to apply, i.e., they apply instantaneously no matter how many they are, while rules take time one. In terms of performance, rules are slightly slower in Maude because they require more infrastructure to be maintained in terms, but that should not be the deciding factor when choosing whether a sentence should be an equation or a rule. One typical criterion for deciding what is an equation and what is a rule is that computations performed with the former are meant to be deterministic, while rewrite rules can lead to non-deterministic behaviors (like our two rules above).

One of Maude’s major strengths, in addition to its efficient support for rewriting modulo ACI, is its capability to perform reachability analysis in conditions of rules. Consider, for example, the following extension of our vending machine model ($S$ and $S'$ are variables of sort State):

\[\text{op both? : State} \rightarrow \text{Bool} .\]
\[\text{crl both?}(S) \Rightarrow \text{true if } S \Rightarrow \text{coffee tea } S' .\]
The conditional rule rewrites \texttt{both?}(S) to \texttt{true} when \( S \) rewrites to at least one coffee and one tea. However, to check the condition, exhaustive search may be needed. Otherwise, wrong normal forms may be reported.

\[
\text{Maude}\geq \text{rew both?}(q q q q q q q q) .
\]
\[
\text{result } \text{Bool}: \text{true}
\]

Maude correctly reported \texttt{true} above; however, without search in the condition one could have wrongly reported the term stuck, for example if one would have bought two coffees from the two dollars in the condition instead of one coffee and one tea.

As expected, the \texttt{search} command also works with conditional rules:

\[
\text{Maude}\geq \text{search both?}(q q q q q q q q) =>^* \text{B:Bool} .
\]
\[
\text{Solution 1 (state 0)}
\]
\[
\text{B:Bool} \rightarrow \text{both?}(q q q q q q q q)
\]
\[
\text{Solution 2 (state 1)}
\]
\[
\text{B:Bool} \rightarrow \text{true}
\]
\[
\text{Solution 3 (state 2)}
\]
\[
\text{B:Bool} \rightarrow \text{both?}(q q q q q q q q)
\]
\[
\text{Solution 4 (state 3)}
\]
\[
\text{B:Bool} \rightarrow \text{false}
\]
\[
\text{Solution 5 (state 4)}
\]
\[
\text{B:Bool} \rightarrow \text{both?}(q q q q q q q q)
\]
\[
\text{Solution 6 (state 5)}
\]
\[
\text{B:Bool} \rightarrow \text{false}
\]
\[
\text{Solution 7 (state 6)}
\]
\[
\text{B:Bool} \rightarrow \text{false}
\]

Interestingly, note that many other solutions besides \texttt{true} have been reported above, some of them in normal form (use \texttt{!} instead of \texttt{*} in the search command to see only those in normal form). That is because the rules of the original vending machine applied inside the argument of \texttt{both?}, which is something that we did not intend to happen when we wrote the conditional rule above. To prohibit rewrite rules from applying in some arguments of an operation, we have to use the \texttt{frozen} operator attribute with corresponding arguments:

\[
\text{op both? : State -> Bool [frozen(1)]} .
\]

Unlike the \texttt{strat} attribute which gives permission to reductions inside operator arguments, the \texttt{frozen} attribute takes permission to rewrites inside operator arguments. The \texttt{strat} attribute only works with equations and is ignored by rewrite rules, while the \texttt{frozen} attribute only works with rewrite rules and is ignored by equations. For example, the above search command still reports two solutions:

\[
\text{Maude}\geq \text{search both?}(q q q q q q q q) =>^* \text{B:Bool} .
\]
\[
\text{Solution 1 (state 0)}
\]
\[
\text{B:Bool} \rightarrow \text{both?}(q q q q q q q q)
\]
\[
\text{Solution 2 (state 1)}
\]
\[
\text{B:Bool} \rightarrow \text{true}
\]

The first solution is an artifact of \texttt{frozen} allowing the equation to apply within the argument of \texttt{both?}. If one wants to prohibit both rules and equations, then one should use both \texttt{frozen} and \texttt{strat} attributes:

\[
\text{op both? : State -> Bool [frozen(1) strat(0)]} .
\]

Rules can have multiple conditions and the conditions can share variables, e.g., \((I, I') \text{ of sort Item})

\[
\text{op solve : State -> State [frozen(1) strat(0)]} .
\]
\[
\text{crl solve}(S) => S' \text{ if } S \Rightarrow I I I S' \land S' \Rightarrow I' I' I' I' S' .
\]
The rule above says that \texttt{solve}(S) rewrites to $S'$ when $S'$ is a rest that can be obtained from $S$ both after buying three identical items and after buying four identical items. Common sense tells us that the three identical items must be coffee and the four identical items must be tea, and that $S'$ is $S$ minus the three dollars spent to buy any of these groups of identical items. But Maude does not have this common sense, it needs to search. Specifically, it will do search in the first condition until three identical items are reached, then it does search in the second condition until four identical items are found with the same rest $S'$ as in the first search; if this is not possible, then it backtracks and searches for another match of three identical items in the first rule, and so on and so forth until the entire state-space is exhausted (if finite, otherwise possibly forever).

Interestingly, although either three coffees or four teas cost three dollars, we cannot buy each of these with three dollars:

\begin{verbatim}
Maude> search solve($ $ $) =>! S .
Solution 1 (state 0)
S --> solve($ $ $)
\end{verbatim}

The term \texttt{solve}($ $ $) is in normal form because there is no way to satisfy the second condition of the conditional rule above. However, if we have one quarter in addition to the three dollars, then we can satisfy the second condition of the rule, too, because we can first buy three teas getting three quarters back, which together with the additional quarter can be changed into one dollar, which gives us one more tea and a quarter back. So from three dollars and a quarter we can buy either three coffees or four teas, with a quarter rest:

\begin{verbatim}
Maude> rew solve($ $ $ q) .
result Coin: q
Maude> search solve($ $ $ q) =>! S .
Solution 1 (state 1)
S --> q
\end{verbatim}

Therefore, both the rewrite and the search commands above have solved the double condition of the conditional rule. Moreover, the search command tells us that there is only one solution to the conditional rule’s constraints.

When giving semantics to programming languages using rewrite logic, conditional rules will be used to reduce the semantic task associated to a language construct to similar semantic tasks associated to its arguments. Since some language constructs have a non-deterministic behavior, the search capability of Maude has a crucial role. In spite of the strength and elegance of Maude’s conditional rules, note, however, that they are quite expensive to execute. Indeed, due to the combined variable constraints resulting from the various conditions, in the worst case there is no way to avoid an exhaustive search of the entire state-space in rules’ conditions. Additionally, each rule used in the search-space of a condition can itself be conditional, which may itself require an exhaustive search to solve its condition, and so on and so forth. All this nested search process is clearly expensive. It is therefore highly recommended to avoid conditional rules whenever possible. As seen in Chapter 3, some semantic styles cannot avoid the use of conditional rules, but others can.

**Turing Machines in Maude**

Section 2.5.3 showed how to faithfully represent Turing machines in rewrite logic, in a way that any computational step in the original Turing machine corresponds to precisely one rewrite step in its (rewrite theory) representation and vice versa. Here we show how such rewrite theories can be defined and executed in Maude. Since the Turing machine transitions are represented as rewrite rules and not as equations, non-deterministic Turing machines can be represented as well following the same approach and without any additional complexity. Following the model in Section 2.5.3 we first discuss the representation based on lazy reduction strategies and then the unrestricted representation. For the lazy one, the idea is to define the infinite tape of zeros lazily, as we did when we defined the various kinds of streams above:
mod TAPE is  
sorts Cell Tape .  
ops 0 1 : -> Cell .  
op _-_ : Cell Tape -> Tape [strat(1 0)] .  
op zeros : -> Tape .  
eq zeros = 0 : zeros .  
endm

Thanks to the strat(1 0) reduction strategy of the _-_ construct above, the expanding equation of zeros only applies when zeros is on a position different from the tail of a stream/tape; in particular, it cannot be applied to further expand the zeros in its RHS.

Now we can define any Turing machine \( M \) in Maude using the approach in Figure 2.10 by importing TAPE, defining operations “q : Tape Tape -> Tape” for all \( q \in Q \), and adding all the rules corresponding to the Turing machine’s transition function (as explained in Figure 2.10). For example, the Maude representation of the Turing machine in Figure 2.1 that computes the successor function is:

mod TURING-MACHINE-SUCC is including TAPE .  
sort Configuration .  
ops qs qh q1 q2 : Tape Tape -> Configuration .  
var L R : Tape . var B : Cell .  
rl qs(L, 0 : R) => q1(0 : L, R) .  
rl q1(L, 0 : R) => q2(L, 1 : R) .  
rl q1(B : L, 1 : R) => q2(L, B : 1 : R) .  
rl q2(L, 0 : R) => qh(L, 0 : R) .  
rl q2(L, B : 1 : R) => q2(L, B : 1 : R) .  
endm

Now we can “execute” the Turing machine using Maude’s rewriting:

Maude> rew qs(zeros, 0 : 1 : zeros) .  
result Configuration: qh(0 : zeros, 0 : 1 : 1 : zeros)  
Maude> rew qs(zeros, 0 : 1 : 1 : 1 : 1 : zeros) .  
result Configuration: qh(0 : zeros, 0 : 1 : 1 : 1 : 1 : 1 : zeros)

Recall from Section 2.2.1 that a natural number input \( n \) is encoded as \( n + 1 \) bits of 1 following the start cell where the head of the machine is initially, which holds the bit 0. So the first rewrite command above rewrites the natural number 0 to its successor 1, while the second rewrites 3 to 4. Note that the self-expanding equation of zeros is not applied backwards, so resulting streams/tapes of the form \( \emptyset : zeros \) are not compacted back into zeros. One needs specific equations to do so, which we do not show here (see Exercise 34).

Non-deterministic Turing machines can be defined equally easily. For example, the machine non-deterministically choosing to yield the successor or not when it reaches the end of the input, also discussed in Section 2.5.3 can be obtained by adding the rule

\[
rl \ q1(B : L, 0 : R) \Rightarrow q2(L, B : \emptyset : R) .
\]

Of course, search is needed now in order to explore all the non-deterministic behaviors:

Maude> search qs(zeros, 0 : 1 : 1 : 1 : 1 : zeros) =>! C:Configuration .  
Solution 1 (state 16)  
C:Configuration --> qh(0 : zeros, 0 : 1 : 1 : 1 : 0 : zeros)  
Solution 2 (state 18)  
C:Configuration --> qh(0 : zeros, 0 : 1 : 1 : 1 : 1 : 1 : zeros)
The unrestricted representation of Turing machines in rewrite logic discussed in Section 2.5.3 (Figure 2.11) can also be easily defined and then executed in Maude. In fact, this unrestricted representation has the advantage that it requires no special support for reduction strategies from the underlying rewrite system, so it should be easily adaptable to other rewrite systems than Maude. Since the self-expanding equation of zeros is not needed anymore, we can now define the tape as a plain algebraic signature:

```plaintext
mod TAPE is
  sorts Cell Tape .
  ops 0 1 : -> Cell .
  op _:_ : Cell Tape -> Tape .
  op zeros : -> Tape .
endm
```

The two new equations in Figure 2.11 can be defined generically as follows, for any Turing machine state:

```plaintext
mod TURING-MACHINE is including TAPE .
  sorts State Configuration .
  op _(_,_,_) : State Tape Tape -> Configuration .
  var S : State .
  var L R : Tape .
  eq S(zeros,R) = S(0 : zeros, R) .
  eq S(L,zeros) = S(L, 0 : zeros) .
endm
```

Particular Turing machines can now be defined by including the module TURING-MACHINE above and adding specific states and rules. For example, here is the one calculating the successor already discussed above:

```plaintext
mod TURING-MACHINE-SUCC is including TURING-MACHINE .
  ops qs qh q1 q2 : -> State .
  var L R : Tape .
  var B : Cell .
  rl qs(L, 0 : R) => q1(0 : L, R) .
  rl q1(L, 0 : R) => q2(L, 1 : R) .
  rl q1(L, 1 : R) => q1(1 : L, R) .
  rl q2(L, 0 : R) => qh(L, 0 : R) .
  rl q2(B : L, 1 : R) => q2(L, B : 1 : R) .
endm
```

The Post Correspondence Problem in Maude

We here show how to define in Maude the rewrite theory in Section 2.5.3 which allows to reduce the Post correspondence problem to rewrite logic reachability. We define strings as AI sequences of symbols, but for output reasons we prefer tiles to be triples instead of just pairs of strings, where the additional string acts as the label of the tile. To easily distinguish labels from each other, we prefer to technically work with strings of natural numbers instead of bits, although we will only use the numbers 0 and 1 in strings not meant to serve as labels. The module below is self-explanatory:

```plaintext
mod PCP is including NAT .
  sorts Symbol String .
  subsort Nat < Symbol < String .
  sorts Tile Tiles .
  subsort Tile < Tiles .
  op . : -> String .  --- empty string
  op __ : String String -> String [assoc id: .] .  --- string concatenation
  op _[.._] : String String String -> Tile [prec 3] .  --- first string is the label
  op _[..] : Tiles Tiles -> Tiles [assoc comm] .  --- concatenation of tiles
  var L L' R R' S S' : String .
  rl L[R,S] L'[R',S'] => L[R,S] L'[R',S'] (L L')(R R', S S') .  --- the only rule
endm
```
So a tile has the form \( \text{label}[^{\alpha,\beta}] \), where \( \text{label} \) is the label of the tile and \( \alpha \) and \( \beta \) are the tile’s two strings. The unique rule matches any two tiles and adds their concatenation to the pool, without removing them. This way, we can start with any set of tiles and eventually reach any combination of them. Obviously this rewrite theory does not terminate. Indeed, if we attempt to execute the command

\[
\text{Maude}> \text{rewrite} \ 1[^{0,1\ 0\ 0}] \ 2[^{0\ 1, \ 0\ 0}] \ 3[^{1\ 1\ 0, \ 1\ 1}] .
\]

then Maude will rewrite forever; in fact, Maude will eventually run out of memory, because each rule application adds one more tile to the pool (and tiles can repeat). The Maude flag \texttt{trace} can be used to see what rewrite steps are applied and is typically used during debugging. For example,

\[
\begin{align*}
\text{Maude}> & \text{set trace on} . \\
\text{Maude}> & \text{rewrite} \ 1[^{0,1\ 0\ 0}] \ 2[^{0\ 1, \ 0\ 0}] \ 3[^{1\ 1\ 0, \ 1\ 1}] .
\end{align*}
\]

outputs an infinite trace starting as follows:

\[
*************** \text{ rule} \\
rl \ L[R,S] \ L'[R',S'] \Rightarrow \ (L'[R',S'] \ (L \ L')[R \ R',S \ S']) \ L[R,S] . \\
L \rightarrow \ 1 \\
R \rightarrow \ 0 \\
S \rightarrow \ 1 \ 0 \ 0 \\
L' \rightarrow \ 2 \\
R' \rightarrow \ 0 \ 1 \\
S' \rightarrow \ 0 \ 0 \\
1[^{0,1\ 0\ 0}] \ 2[^{0\ 1, \ 0\ 0}] \ 3[^{1\ 1\ 0, \ 1\ 1}] \\
\rightarrow \ 3[^{1\ 1\ 0, \ 1\ 1}] \ (2[^{0\ 1, \ 0\ 0}] \ (1 \ 2)[^{0\ 0 \ 1, \ (1 \ 0 \ 0) \ 0 \ 0}]) \ 1[^{0,1\ 0\ 0}] \\
*************** \text{ rule} \\
rl \ L[R,S] \ L'[R',S'] \Rightarrow \ (L'[R',S'] \ (L \ L')[R \ R',S \ S']) \ L[R,S] . \\
L \rightarrow \ 1 \\
R \rightarrow \ 0 \\
S \rightarrow \ 1 \ 0 \ 0 \\
L' \rightarrow \ 2 \\
R' \rightarrow \ 0 \ 1 \\
S' \rightarrow \ 0 \ 0 \\
1[^{0,1\ 0\ 0}] \ 2[^{0\ 1, \ 0\ 0}] \ 3[^{1\ 1\ 0, \ 1\ 1}] \ (1 \ 2)[^{0\ 0 \ 1, \ 1 \ 0 \ 0 \ 0 \ 0 \ 0}]) \\
\rightarrow \ (3[^{1\ 1\ 0, \ 1\ 1}] \ (1 \ 2)[^{0\ 0 \ 1, \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0}]) \ (2[^{0\ 1, \ 0\ 0}] \ (1 \ 2)[^{0\ 0 \ 1, \ (1 \ 0 \ 0) \ 0 \ 0}]) \ 1[^{0,1\ 0\ 0}] \\
\ldots
\]

Each step in the trace shows a rule instance and the resulting term (Maude may re-arrange the AC terms). Here it just happens that Maude chooses the same first two tiles to combine at each step.

Maude also allows to pass an optional number to the rewrite command, indicating how many rewrite steps are desired. For example:

\[
\begin{align*}
\text{Maude}> & \text{rewrite}[2] \ 1[^{0,1\ 0\ 0}] \ 2[^{0\ 1, \ 0\ 0}] \ 3[^{1\ 1\ 0, \ 1\ 1}] . \\
\text{result Tiles:} & \ 1[^{0,1\ 0\ 0}] \ 2[^{0\ 1, \ 0\ 0}] \ 3[^{1\ 1\ 0, \ 1\ 1}] \ (1 \ 2)[^{0\ 0 \ 1, \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0}]) \ (1 \ 2)[^{0\ 0 \ 1, \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0}]) \\
\end{align*}
\]

One should be careful when using this option with conditional rewrite rules, because only the top-level applications of rules are counted. The rules used to solve the conditions are \textit{not} counted. It is therefore possible that the command \texttt{rewrite[1]} does not terminate, namely when the condition of a matching conditional rule does not terminate.

Consequently, we can only use the \texttt{search} command here to solve the PCP. Specifically, we should search for patterns containing a tile of the form \( \text{label}[^{\gamma,\gamma}] \). Any term of sort \texttt{Tiles} matching such a pattern
indicates that a successful combination of the original tiles has been found; moreover, *label* will be bound to the desired sequence of labels of the combined tiles and $\gamma$ to the combined string. For example, the following shows that the specific PCP problem mentioned in Section 2.2.2 is solvable, as well as a solution:

Maude> search[1] 1[0, 1 0 0] 2[0 1, 0 0] 3[1 1 0, 1 1] =>* L[S,S] Ts:Tiles .
L --> 3 2 3 1
S --> 1 1 0 0 1 1 1 0 0
Ts:Tiles --> 1[0,1 0 0] 2[0 1,0 0] 3[1 1 0,1 1] (2 3)[0 1 1 0,0 0 1 1] ...

It is important to use *search*[1] above, because otherwise Maude will continue to search for all solutions and thus will never terminate. The option [1] tells Maude to stop searching after finding one solution.

Since the PCP problem is undecidable, we can conclude that Maude’s search is undecidable in general (which is not a surprise, but we now have a formal proof).

### 2.5.7 Exercises

**Exercise 27.** Eliminate the two generic equations in Figure 2.11 as discussed below Theorem 11 and prove that the resulting rewrite theory is confluent and captures the computation in $M$ faithfully, step-for-step.

**Exercise 28.** Prove that any correct implementation of *PEANO-NAT* (Section 2.5.6) should satisfy the property

```
plus(succ(succ(zero))), succ(succ(succ(zero)))) =
plus(succ(succ(succ(succ(zero))))), succ(zero)).
```

**Exercise 29.** Rewrite *PEANO-NAT* and *PEANO-NAT* (Section 2.5.6) using Maude’s mixfix notation for operations. What happens if we try to reduce an expression containing both _+_ and _*_ without parentheses?

**Exercise 30.** Define a Maude module called *INT-SET* specifying sets of integers with membership, union, intersection, difference (elements in the first set but not in the second), and symmetric difference (elements in any of the two sets but not in the other).

**Exercise 31.** Define the *search* operation in module *SEARCH* (Section 2.5.6) with only two unconditional equations, using the built-in *if_then_else_fi*.

**Exercise 32.** Recall module *FLATTEN* (Section 2.5.6) which defines and infix traversal operation on binary trees. Do the same for prefix and for postfix traversals.

**Exercise 33.** Write a Maude module that uses binary trees as defined in module *TREE* (Section 2.5.6) to sort lists of integers. You should define an operation *btsort* : *IntList* -> *IntList*, which sorts the argument list of integers (like the *isort* operation in module *ISORT* in Section 2.5.6). In order to define *btsort*, define another operation, *bt-insert* : *IntList* *Tree* -> *Tree*, which inserts each integer in the list at its place in the tree, and also use the *flatten* operation already defined in module *FLATTEN*.

**Exercise 34.** When executing Turing machines in Maude as shown in Section 2.5.6, we obtain final configurations which contain (sub)streams/tapes of the form $\theta : \text{zeros}$. While these are semantically equal to *zeros*, they decrease the readability of the final Turing machine configurations. Add generic equations to canonicalize the final configurations by iteratively replacing each sub-term $\theta : \text{zeros}$ with *zeros*. **Hint:** A special reduction strategy may be needed for the operation *qh* (or the configuration construct, respectively), to inhibit the application of the self-expanding equation of *zeros*.

82
**Exercise 35.** Define in Maude the Turing machines corresponding to the addition, the multiplication, and the power operations on natural numbers in Exercise 12. Do it using three different approaches: (1) using the infinite stream zeros, following the representation in Figure 2.10; (2) without using infinite streams but mimicking them with the by-need expansion using the two equations in Figure 2.11; (3) without any equations, following the style suggested right after Theorem 11 at the expense of adding more rules.

**Exercise 36.** Use the Maude definition of the Post correspondence problem in Section 2.5.6 to calculate the least common multiplier of two natural numbers. The idea here is to create an appropriate set of tiles from the two numbers so that the solution to the search command contains the desired least common multiplier.
Talk about kinds and MEL? We currently need them in rsec (contextual representation of syntax is a kind), but maybe we can eliminate them from there, too.

Discuss the LTL model checker? If yes, the dining philosophers example with rewrite rules.

2.5.8 Notes

Maude’s roots go back to Clear (Edinburgh) in 1970s and to OBJ (Stanford, Oxford) in 1980s. There are many other languages in the same family, such as CafeOBJ (Japan), BOBJ (San Diego, California), ELAN (France).

Maude is currently being developed at Stanford Research Institute and at the University of Illinois at Urbana-Champaign. You can find more about Maude, including a manual and many papers and examples, on its webpage at http://maude.cs.uiuc.edu.

Salvador with strategies
2.6 Complete Partial Orders and the Fixed-Point Theorem

This section introduces a fixed-point theorem for complete partial orders. Our main use of this theorem is to give denotational semantics to iterative (in Section 3.4) and to recursive (in Section 4.7) language constructs. Complete partial orders with bottom elements are at the core of both domain theory and denotational semantics, often identified with the notion of “domain” itself.

2.6.1 Posets, Upper Bounds and Least Upper Bounds

Here we recall basic notions of partial and total orders, such as upper bounds and least upper bounds, and discuss several examples and counter-examples.

Definition 14. A partial order, or a poset (from partial order set) \((D, \sqsubseteq)\) consists of a set \(D\) and a binary relation \(\sqsubseteq\) on \(D\), written as an infix operation, which is

- **reflexive**, that is, \(x \sqsubseteq x\) for any \(x \in D\);
- **transitive**, that is, for any \(x, y, z \in D\), \(x \sqsubseteq y\) and \(y \sqsubseteq z\) imply \(x \sqsubseteq z\); and
- **anti-symmetric**, that is, if \(x \sqsubseteq y\) and \(y \sqsubseteq x\) for some \(x, y \in D\) then \(x = y\).

A partial order \((D, \sqsubseteq)\) is called a total order when \(x \sqsubseteq y\) or \(y \sqsubseteq x\) for any \(x, y \in D\).

Here are some examples of partial or total orders:

- \((\mathcal{P}(S), \subseteq)\) is a partial order, where \(\mathcal{P}(S)\) is the powerset of a set \(S\), that is, the set of subsets of \(S\).
- \((\text{Nat}, \leq)\), the set of natural numbers ordered by “less than or equal to”, is a total order.
- \((\text{Nat}, \geq)\), the set of natural numbers ordered by “larger than or equal to”, is a total order.
- \((\text{Nat} \cup \{\infty\}, \leq)\), the naturals plus infinity, where infinity is larger than any number, is a total order.
- \((\text{Int}, \leq)\), the set of integer numbers, is a total order.
- \((\text{Real}, \leq)\), the set of real numbers, is a total order.
- \((S, =)\), a flat set \(S\) where the only partial ordering is the identity, is a partial order.
- \((S \cup \{\bot\}, \leq_{S}^{\bot})\), a set \(S\) extended with a bottom element \(\bot\) with the property that for any \(a, b \in S \cup \{\bot\}\), \(a \leq_{S}^{\bot} b\) if and only if \(a = b\) or \(a = \bot\), is a partial order. Such extensions of sets with bottom elements are common in denotational semantics (Section 3.4), where the intuition for the \(\bot\) is undefined. They are commonly written more simply as \(S_{\bot}\) and are often called primitive or flat domains.
- A partial order which is particularly important for denotational semantics (Section 3.4) is \((A \rightarrow B, \leq)\), the set of partial functions from \(A\) to \(B\) partially ordered by the informativeness relation \(\leq\): given partial functions \(f, g : A \rightarrow B\), \(f\) is “less informative than or as informative as” \(g\), written \(f \leq g\), iff for any \(a \in A\), it is either the case that \(f(a)\) is not defined or both \(f(a)\) and \(g(a)\) are defined and \(f(a) = g(a)\).
• If \((S_1, \leq_1)\) and \((S_2, \leq_2)\) are partial or total orders and \(S_1 \cap S_2 = \emptyset\) then \((S_1 \cup S_2, \leq_1 \cup \leq_2)\) is a partial order, where \(a (\leq_1 \cup \leq_2)b\) if and only if there is some \(i \in \{1, 2\}\) such that \(a, b \in S_i\) and \(a \leq_i b\). This union partial order is typically not a total order; it is a total order only when \((S_1, \leq_1)\) is total and \(S_2 = \emptyset\), or the other way around. More generally, if \(\{(S_i, \leq_i)\}_{i \in I}\) is a family of partial orders such that \(S_i \cap S_j = \emptyset\) for any \(i \neq j \in I\), then \(\bigcup_{i \in I}(S_i, \leq_i) \overset{\text{def}}{=} (\bigcup_{i \in I} S_i, \bigcup_{i \in I} \leq_i)\) is a partial order, where \(a (\bigcup_{i \in I} \leq_i) b\) if and only if there is some \(i \in I\) such that \(a, b \in S_i\) and \(a \leq_i b\).

• If \(\{(S_i, \leq_i)\}_{i \in I}\) is a family of partial orders then \(\prod_{i \in I}(S_i, \leq_i) \overset{\text{def}}{=} (\prod_{i \in I} S_i, \prod_{i \in I} \leq_i)\) is also a partial order, where \(\{a_i\}_{i \in I} (\prod_{i \in I} \leq_i) \{b_i\}_{i \in I}\) iff \(a_i \leq_i b_i\) for all \(i \in I\) (i.e., the product partial order is defined component-wise).

**Definition 15.** Given partial order \((D, \sqsubseteq)\) and a set of elements \(X \subseteq D\), an element \(p \in D\) is called an upper bound of \(X\) iff \(x \sqsubseteq p\) for any \(x \in X\). Furthermore, \(p \in D\) is called the least upper bound (LUB) of \(X\), written \(\sqcup X\), iff \(p\) is an upper bound and for any other upper bound \(q\) of \(X\) it is the case that \(p \sqsubseteq q\).

Upper bounds and least upper bounds may not always exist. For example, if \(D = X = \{x, y\}\) and \(\sqsubseteq\) is the identity relation, then \(X\) has no upper bounds. Least upper bounds may not exist even though upper bounds exist. For example, if \(D = \{a, b, c, d, e\}\) and \(\sqsubseteq\) is defined by \(a \sqsubseteq c, a \sqsubseteq d, b \sqsubseteq c, b \sqsubseteq d, c \sqsubseteq e, d \sqsubseteq e\), then any subset of \(D\) admits upper bounds, but the set \(\{a, b\}\) does not have a LUB. Due to the anti-symmetry property, least upper bounds are unique when they exist, which justifies the notation \(\sqcup X\) in Definition 15.

As an analogy with mathematical analysis, we can think of the LUB of a set of elements as their limit. For example, in \((\text{Real}, \leq)\) the LUB of the set \(\{n/(n+1) \mid n \in \text{Nat}\}\) coincides with its limit in the mathematical analysis sense, namely 1. Note that \((\{0, 1\}, \leq)\) does not admit LUBs for all its subsets, e.g., \(\{n/(n+1) \mid n \in \text{Nat}\}\) does not have a LUB. One might think that this happens because \((\{0, 1\}, \leq)\) does not even admit upper bounds for all its subsets. If we add the interval \([4, 5]\) to the set above, for example, then the resulting (total) order admits upper bounds for any of its subsets (e.g., 5 is such an upper bound) but the set \(\{n/(n+1) \mid n \in \text{Nat}\}\) still does not have a LUB. However, if we add the interval \([4, 5]\) to the original interval \([0, 1]\) then the resulting (total) order admits LUBs for all its subsets; for example, the LUB of \(\{n/(n+1) \mid n \in \text{Nat}\}\) is 4.

### 2.6.2 Complete Partial Orders

In the decimal representation of real numbers, completeness means that any infinite string of decimal digits is actually the decimal representation for some real number. In mathematical analysis in general, a set of real numbers is complete iff the limit of any converging sequence of real numbers is also in the set. For example, the interval \([0, 1]\) open in 1 is not complete, because the sequence \(1 - 1/n\) for \(n > 0\) converges to 1 but 1 is not in the set. On the other hand, the interval \([0, 1]\) is complete. Many particular variants of completeness that appear in the literature are in fact instances a general notion or completeness for partial orders.

**Definition 16.** Given a poset \((D, \sqsubseteq)\), a **chain** in \(D\) is an infinite sequence \(d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots\) of elements in \(D\), also written using the set notation as \(\{d_n \mid n \in \text{Nat}\}\). Such a chain is called **stationary** when there is some \(n \in \text{Nat}\) such that \(d_m = d_{m+1}\) for all \(m \geq n\).

If \(D\) is finite then any chain is stationary. More generally, if for a given \(x \in D\) there are only finitely many elements \(y \in D\) such that \(x \sqsubseteq y\), then any chain containing \(x\) is stationary.

**Definition 17.** A poset \((D, \sqsubseteq)\) is called a **complete partial order (CPO)** iff any of its chains has a LUB. \((D, \sqsubseteq)\) is said to have **bottom** iff it has a minimal element. Such an element is typically written \(\bot\), and the poset
with bottom \( \bot \) is written \((D, \sqsubseteq, \bot)\). CPOs with bottom are also called **bottomed complete partial orders**. If \( \{d_n \mid n \in \text{Nat}\} \) is a chain in \((D, \sqsubseteq)\), then we also let \( \bigsqcup_{n \in \text{Nat}} d_n \), or simply \( \sqcup d_n \), denote its LUB, \( \sqcup \{d_n \mid n \in \text{Nat}\} \).

Let us follow up on the examples underneath Definition 14 and discuss which of them are CPOs with botom and which are not:

- \((\mathcal{P}(S), \subseteq, \emptyset)\) is a bottomed CPO.
- \((\text{Nat}, \leq)\) has bottom 0 but is not complete: the chain \(0 \leq 1 \leq 2 \leq \cdots \leq n \leq \cdots\) has no upper bound.
- \((\text{Nat}, \geq)\) is a CPO but has no bottom.
- \((\text{Nat} \cup \{\infty\}, \leq, 0)\) is a CPO with bottom 0: it is a CPO because any chain is either stationary, in which case its LUB is obvious, or is unbounded by any natural number, in which case \(\infty\) is its LUB.
- \((\text{Int}, \leq)\) is not a CPO and has no bottom.
- \((S, =)\) is always a CPO, and it has bottom if and only if \(S\) has only one element.
- \((S \cup \{\bot\}, \leq, \bot)\) is a bottomed CPO. It is common to use the same notation \(S_\bot\), used for the partial order \((S \cup \{\bot\}, \leq_\bot)\), to also denote its corresponding bottomed CPO \((S \cup \{\bot\}, \leq_\bot, \bot)\). It is interesting and particularly important in denotational semantics to note that the primitive domain \(S_\bot\) is equivalent to the set of partial functions \(\{\ast\} \to S\), where \(\{\ast\}\) is the singleton set.
- \((A \to B, \leq, \bot)\), the set of partial functions \(A \to B\) ordered by the informativeness relation \(\leq\), is a CPO with bottom \(\bot : A \to B\), the partial function which is undefined in each element of \(A\).
- If \(\{(S_i, \leq_i, \bot)\}_{i \in I}\) is a family of bottomed CPOs with same bottom \(\bot\) such that \(S_i \cap S_j = \{\bot\}\) for any \(i \neq j \in I\), then \(\bigcup_{i \in I}(S_i, \leq_i, \bot)\) is also a bottomed CPO.
- If \(\{(S_i, \leq_i)\}_{i \in I}\) is a family of bottomed CPOs then \(\prod_{i \in I}(S_i, \leq_i)\) is also a bottomed CPO.

All the CPOs of interest discussed in this book have a bottom element and, with a few minor exceptions in the next section, all the results that we prove for CPOs require bottom elements. Consequently, unless otherwise explicitly stated, we assume that all CPOs are bottomed CPOs.

### 2.6.3 Monotone and Continuous Functions

The same way various notions of completeness encountered in the literature are instances of the general notion of complete partial orders in Section 2.6.2 various particular notions of monotone and of continuous functions are instances of homonymous notions defined generally for partial orders and CPOs.

**Definition 18.** If \((D, \sqsubseteq)\) and \((D', \sqsubseteq')\) are two posets and \(F : D \to D'\) is a function, then \(F\) is called **monotone** if and only if \(F(x) \sqsubseteq' F(y)\) for any \(x, y \in D\) with \(x \sqsubseteq y\). If \(F\) is monotone, then we simply write \(F : (D, \sqsubseteq) \to (D', \sqsubseteq')\). Let \(\text{Mon}(D, \sqsubseteq, (D', \sqsubseteq'))\) denote the set of monotone functions from \((D, \sqsubseteq)\) to \((D', \sqsubseteq')\). Monotone functions preserve chains, that is, \(\{F(d_n) \mid n \in \text{Nat}\}\) is a chain in \((D', \sqsubseteq')\) whenever \(\{d_n \mid n \in \text{Nat}\}\) is a chain in \((D, \sqsubseteq)\). Moreover, if \((D, \sqsubseteq)\) and \((D', \sqsubseteq')\) are CPOs then for any chain \(\{d_n \mid n \in \text{Nat}\}\) in \((D, \sqsubseteq)\), we have \(\sqcup F(d_n) \sqsubseteq' F(\sqcup d_n)\). Indeed, since \(F\) is monotone and since \(d_n \sqsubseteq \sqcup d_n\) for each \(n \in \text{Nat}\), it follows
that \( \mathcal{F}(d_n) \subseteq \mathcal{F}(\sqcup d_n) \) for each \( n \in \text{Nat} \). Therefore, \( \mathcal{F}(\sqcup d_n) \) is an upper bound for the chain \( \{ \mathcal{F}(d_n) \mid n \in \text{Nat} \} \). The rest follows because \( \sqcup \mathcal{F}(d_n) \) is the LUB of \( \{ \mathcal{F}(d_n) \mid n \in \text{Nat} \} \).

Note that \( \mathcal{F}(\sqcup d_n) \subseteq \sqcup \mathcal{F}(d_n) \) does not hold in general. Let, for example, \( (D, \sqsubseteq) \) be \( (\text{Nat} \cup \{ \infty \}, \leq) \), \( (D', \sqsubseteq') \) be \( (0, \infty), 0 \leq \infty \), and \( \mathcal{F} \) be the monotone function taking any natural number to 0 and \( \infty \) to \( \infty \). For the chain \( \{ n \mid n \in \text{Nat} \} \), note that \( \sqcup n = \infty \), so \( \mathcal{F}(\sqcup n) = \infty \). On the other hand, the chain \( \{ \mathcal{F}(n) \mid n \in \text{Nat} \} \) is stationary in 0, so \( \sqcup \mathcal{F}(n) = 0 \). Therefore, \( \mathcal{F}(\sqcup n) = \infty \not\leq 0 = \sqcup \mathcal{F}(n) \).

Recall that one can think of the LUB of a chain as the limit of that chain. The following definition is inspired from the analogous notion of continuous function in mathematical analysis, which is characterized by the property of preserving limits:

**Definition 19.** Let \( (D, \sqsubseteq) \) and \( (D', \sqsubseteq') \) be CPOs. A monotone function \( \mathcal{F} : (D, \sqsubseteq) \to (D', \sqsubseteq') \) is **continuous** iff \( \mathcal{F}(\sqcup d_n) \subseteq' \sqcup \mathcal{F}(d_n) \), which is equivalent to \( \sqcup \mathcal{F}(d_n) = \mathcal{F}(\sqcup d_n) \), for any chain \( \{ d_n \mid n \in \text{Nat} \} \) in \( (D, \sqsubseteq) \). Let \( \text{Cont}(D, \sqsubseteq) \) denote the set of continuous functions from \( (D, \sqsubseteq) \) to \( (D', \sqsubseteq') \). This notation extends seamlessly to bottomed CPO: \( \text{Cont}(D, \sqsubseteq, \perp) \).

If we let \( S_{\perp} \) denote the bottomed CPO extension of the set \( S \) as discussed in Section 2.6.2, then it can be shown that there is a bijective correspondence between the set of partial functions in \( A \to B \) and the set of continuous (total) functions in \( \text{Cont}(A_{\perp}, B_{\perp}) \) (see Exercises 37 and 38).

Note that continuous functions in \( \text{Cont}(D, \sqsubseteq, \perp) \) need not take \( \perp \) to \( \perp' \) in general. In fact, as seen shortly in Section 2.6.4, the interesting continuous functions do not have that property.

**Proposition 12.** \( \text{Cont}(D, \sqsubseteq, \perp) \) can be endowed with a bottomed CPO structure, where

- Its partial order is defined as \( f \leq g \) iff \( f(d) \leq g(d) \) for all \( d \in D \);
- Its bottom element is the continuous function taking each element \( d \in D \) to \( \perp' \).

**Proof.** ...

Recall from Section 2.6.2 that for any sets \( A \) and \( B \), the set of partial functions \( A \to B \) can be organized as a bottomed CPO, \( (A \to B, \leq, \perp) \). Moreover, it can be shown (see Exercise 39) that \( (A \to B, \leq, \perp) \) and \( (\text{Cont}(A_{\perp}, B_{\perp}), \leq, \perp) \) are in fact isomorphic bottomed CPOs. This way, Proposition 12 allows us to associate a bottomed CPO to any higher-order type. For example, the type \( (\text{int} \to \text{int}) \to \text{int} \to \text{bool} \) of functions taking functions from \( \text{Int} \to \text{Int} \) into functions from \( \text{Int} \to \text{Bool} \) can be associated the bottomed CPO \( \text{Cont}(\text{Int}_{\perp}, \text{Int}_{\perp}), \text{Cont}(\text{Int}_{\perp}, \text{Bool}_{\perp}) \). This uniform view of types as CPOs will allows us in Section 4.7 to use the fixed-point theorem discussed next to elegantly give denotational semantics to higher-order languages with recursive functions of arbitrary types.

Moreover, we can organize all the CPOs above together as one big CPO. Indeed, given any family of bottomed CPOs \( \{ B_i \}_{i \in I} \) (thought of as CPOs corresponding to some arbitrary set of basic types), let \( \mathcal{HO} \) (from higher-order model) be the smallest set closed under the following:

- \( B_i \in \mathcal{HO} \) for any \( i \in I \);
- If \( X, Y \in \mathcal{HO} \) then \( \text{Cont}(X, Y) \in \mathcal{HO} \).

In Section 2.7 we will see that \( \mathcal{HO} \) can be regarded as a cartesian-closed-category. Now let us additionally assume that all CPOs \( B_i \) have the same bottom element \( \perp \) and that \( B_i \cap B_j = \{ \perp \} \) for any \( i, j \in I \). Also, assume that we use the same \( \perp \) element as bottom in all the CPOs \( \text{Cont}(X, Y) \in \mathcal{HO} \). Since the union of CPOs whose pairwise intersection is their common bottom element \( \{ \perp \} \) is also a CPO with bottom \( \{ \perp \} \) (Section 2.6.2), we conclude that the union of all the CPOs in \( \mathcal{HO} \) is also a CPO with bottom \( \{ \perp \} \).
2.6.4 The Fixed-Point Theorem

Any monotone function $F : (D, \sqsubseteq, \bot) \rightarrow (D, \sqsubseteq, \bot)$ defined on a bottomed CPO to itself admits an implicit and important chain, namely $\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \sqsubseteq \cdots \sqsubseteq F^n(\bot) \sqsubseteq \cdots$, where $F^n$ denotes $n$ compositions of $F$ with itself. The next theorem is a key result, which has major implications in many areas of mathematics and computer science, in particular in denotational semantics (Section 3.4):

**Theorem 12. (The fixed-point theorem)** Let $(D, \sqsubseteq, \bot)$ be a bottomed CPO, let $F : (D, \sqsubseteq, \bot) \rightarrow (D, \sqsubseteq, \bot)$ be a continuous function, and let $\text{fix}(F)$ be the LUB of the chain $(F^n(\bot) \mid n \in \text{Nat})$. Then $\text{fix}(F)$ is the least fix-point of $F$.

**Proof.** We first show that $\text{fix}(F)$ is a fix-point of $F$:

$$F(\text{fix}(F)) = F(\bigcup_{n \in \text{Nat}} F^n(\bot)) = \bigcup_{n \in \text{Nat}} F^{n+1}(\bot) = \bigcup_{n \in \text{Nat}} F^n(\bot) = \text{fix}(F).$$

Next we show that $\text{fix}(F)$ is the least fix-point of $F$. Let $d$ be another fix-point of $F$, that is, $F(d) = d$. We can show by induction that $F^n(\bot) \sqsubseteq d$ for any $n \in \text{Nat}$: first note that $F^0(\bot) = \bot \sqsubseteq d$; assume $F^n(\bot) \sqsubseteq d$ for some $n \in \text{Nat}$; since $F$ is monotone, it follows that $F(F^n(\bot)) \sqsubseteq F(d) = d$, that is, $F^{n+1}(\bot) \sqsubseteq d$. Thus $d$ is an upper bound of the chain $(F^n(\bot) \mid n \in \text{Nat})$, so $\text{fix}(F) \sqsubseteq d$. \hfill \Box

Many recursive definitions in mathematics and computer science are given informally, but they are more subtle than they appear to be. The fixed-point theorem can be used to formally argue that such definitions are indeed correct. Consider, for example, the following common definition of the factorial:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \ast f(n - 1) & \text{if } n > 0 \end{cases}$$

How can we know whether such a mathematical object, i.e., a function $f$ satisfying the above property, actually exists and is unique, as tacitly assumed? To see that such a concern is justified, replace “$n \ast f(n - 1)$ if $n > 0$” by “$n \ast f(n - 1)$ if $n > 1$” in the above; now there are infinitely many functions satisfying the above property. Or replace $n \ast f(n - 1)$ by $n \ast f(n + 1)$; now there is no function with the property above. According to the fixed-point theorem, since the function $F$ defined on the set of partial functions $\text{Nat} \rightarrow \text{Nat}$ to itself as

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \ast g(n - 1) & \text{if } n > 0 \text{ and } g(n - 1) \text{ defined} \\ \text{undefined} & \text{if } n > 0 \text{ and } g(n - 1) \text{ undefined} \end{cases}$$

is continuous, it has a least fixed point. We thus can take $f = \text{fix}(F)$, and get

$$f(n) = F(f)(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \ast f(n - 1) & \text{if } n > 0 \text{ and } f(n - 1) \text{ defined} \\ \text{undefined} & \text{if } n > 0 \text{ and } f(n - 1) \text{ undefined} \end{cases}$$

One can easily show (by induction) that $f$ is defined everywhere, so the above is equivalent to the original definition of $f$. Since $f$ is total, it is the unique fixed point of $F$: any fixed-point $f'$ of $F$ obeys $f \preceq f'$, so $f'$ is also defined everywhere and equal to $f$. See also Exercise 40.
Another application of the fixed-point theorem is to show why and how recursively defined languages admit unique solution. Any context-free language over a possibly infinite alphabet, or set of terminals, is the least fixed point of some continuous operator on the power set of the set of words over the given alphabet. Let for instance the alphabet be \( A = \text{Var} \cup \text{Int} \cup \{ +, -, \ast \} \), where \( \text{Int} \) is the set of integers and \( \text{Var} \) is a set of variables, and consider the following context-free grammar giving the syntax for arithmetic expressions:

\[
\text{Exp} ::= \text{Var} \mid \text{Int} \mid \text{Exp} + \text{Exp} \mid -\text{Exp} \mid \text{Exp} \ast \text{Exp}
\]

There are many ways to describe or construct the language of arithmetic expressions corresponding to the grammar above, which we do not discuss here. What we want to highlight is that it can also be described as the least fixed point of the following continuous function, where \( A^+ \) is the set of finite words with letters in \( A \):

\[
F : (\mathcal{P}(A^+), \subseteq, \emptyset) \to (\mathcal{P}(A^+), \subseteq, \emptyset),
\]

\[
F(L) = \text{Var} \cup \text{Int} \cup L_+L \cup \{-\}L \cup L_\ast L.
\]

We used the notation \( L_1L_2 = \{ w_1w_2 \mid w_1 \in L_1, w_2 \in L_2 \} \). Notice that the iterations \( F(\emptyset), F^2(\emptyset), \ldots \) correspond to the one-step, two-steps, \ldots derivations applying the grammar’s productions. See Exercise 41.

We shall later see that the fixed-point theorem above can also be used to give denotational semantics both to iterative (in Section 3.4) and to recursive (in Section 4.7.3) constructs. Fixed-points in general and the theorem above in particular have many applications in both computer science and mathematics.

### 2.6.5 Equational Framework for Fixed-Points

As seen above, fixed-points can be complex mathematical objects, such as functions or infinite sets of words, which may require complex means to define, understand or visualize. In many cases the only option is to experiment with the fixed-point, such as to evaluate it on a given input when it is a function, or to check whether it contains a certain element when it is a set. If one wants to formally reason about or to compute fixed points, then one needs to formally support, explicitly or implicitly, the body of mathematics involved in the definition of the function for which the fixed point is desired. For example, the factorial above requires support for the domain of integers, at least with multiplications and subtraction, as well as support for the domain \( \text{Int} \to \text{Int} \) of partial functions from integers to integers (in order to define \( F \)). Similarly, the CFG language above requires support for sets with union and possibly other operations on them, and for the domain of finite words over a given alphabet, with at least a concatenation operation on them, as well as a combination of the two, where concatenation is extended to sets of words. Even though in theory we assume the entire arsenal of mathematics, in practice we only have a limited fragment of mathematics available for formal definitions of fixed points. In this section we define such a limited framework, but one which is quite common and particularly useful for the application of fixed-points in denotational semantics (Section 3.4).

Figure 2.12 shows a simple equational theory of a minimal framework for formally defining fixed-points. We call it minimal because it only includes support for defining and applying functions, together with a simple extension for defining cases. One will need to add further extensions to this framework in order to support definitions of more complex fixed-points. For defining cases, we assumed that Booleans are already defined. For convenience, from here on we also assume that integers are already defined, together with all the desired operations on both Booleans and integers. To make it clear from which mathematical domain each operation or relational symbol comes, we add the main domain sort as a subscript to the operator or relational symbol. For example, \( +_{\text{Int}} \) is the addition on integers, \( \leq_{\text{Int}} \) is their comparison which evaluates to a Boolean, and \( =_{\text{Int}} \) is their equality test. The functions, their application and their fixed-points have been defined like in untyped call-by-value \( \lambda \)-calculus (see Section 4.5) extended with the recursion \( \mu \) construct (see Section 4.7),
sorts:
\( \text{Var}_{\text{CPO}}, \ CPO \)

subsorts:
\( \text{Var}_{\text{CPO}}, \ \text{Bool} < \ CPO \)

operations:
\[
\begin{align*}
\text{fun}_{\text{CPO}} &: \text{Var}_{\text{CPO}} \times CPO \rightarrow CPO \\
\text{app}_{\text{CPO}} &: CPO \times CPO \rightarrow CPO \\
\text{fix}_{\text{CPO}} &: CPO \rightarrow CPO \\
\text{if}_{\text{CPO}} &: \text{Bool} \times CPO \times CPO \rightarrow CPO
\end{align*}
\]

equations:
\[
\begin{align*}
\text{app}_{\text{CPO}}(\text{fun}_{\text{CPO}} V \rightarrow C, C') &= C[C'/V] \\
\text{fix}_{\text{CPO}}(\text{fun}_{\text{CPO}} V \rightarrow C) &= C[(\text{fix}_{\text{CPO}} \text{fun}_{\text{CPO}} V \rightarrow C)/V] \\
\text{if}_{\text{CPO}}(\text{true}, C, C') &= C \\
\text{if}_{\text{CPO}}(\text{false}, C, C') &= C'
\end{align*}
\]

Figure 2.12: Equational framework for CPOs and fixed-points (assuming Booleans and substitution).

using an appropriate notion of substitution (see Section 4.5.3). How these are defined is not important here, so we refer the interested reader to the above-mentioned sections for details. What is important here is that the equational theory in Figure 2.12 gives us a starting point for formally defining fixed-points.

For simplicity, we considered only one major syntactic category in the equational theory in Figure 2.12, namely \( CPO \), for complete partial orders. Basic CPOs other than that of Booleans \( \text{Bool} \), e.g., the CPO of integers \( \text{Int} \), need to be subsorted to \( CPO \) if one wants to include them. We concentrate on functional domains here, that is, on CPOs whose elements are partial functions, with \( \bot \) the partial function which is undefined everywhere. We followed our naming convention and thus we tagged all the newly introduced operations with \( \text{CPO} \), to distinguish them from possibly homonymous operations from included domains. We implicitly assume that the various domains subsorted to \( CPO \) are responsible for generating undefinedness (e.g., when a division by zero in the domain of integers is attempted) and appropriately propagating it. The four equations in Figure 2.12 are straightforward, capturing the meaning of the four language constructs in terms of substitution; here we assume substitution given, as explained above.

We can now use the constructs in Figure 2.12 to define fixed-points. For example, the factorial function discussed above can be defined as the term below of sort \( CPO \), say \( \text{factorial} \):

\[
\text{fix}_{\text{CPO}} \text{fun}_{\text{CPO}} g \rightarrow \text{fun}_{\text{CPO}} n \rightarrow \text{if}_{\text{CPO}}(n =_{\text{int}} 0, 1, \text{if}_{\text{CPO}}(n >_{\text{int}} 0, n *_{\text{int}} \text{app}_{\text{CPO}}(g, n -_{\text{int}} 1), \bot))
\]

Using the provided equations, one can prove, for example, that \( \text{factorial}(3) = 6 \). The only difference between our definition of factorial above and the fixed-point definition right after Theorem 12 is that we assumed the common mathematical convention that undefinedness is propagated implicitly through the various domain operations, so we did not propagate it explicitly. See Exercise 42 for another example.

The equational theory in Figure 2.12 can be easily defined in rewrite logic systems like Maude, provided that a generic substitution is available. For example, Appendix A.1 shows one way to do it, together with several extensions not discussed here. Once such a Maude definition of the mathematical domain is available, we can use it to calculate and experiment with fixed-points, in particular to execute recursive functions like the factorial above and, more importantly, later (Section 3.4) to execute denotational semantics of programming languages, thus obtaining interpreters for the defined languages directly from their formal semantics.
2.6.6 Exercises

Exercise 37. Recall from Section 2.6.2 that given a set $S$, we let $S_\bot$ denote the poset $(S \cup \{\bot\}, \leq^S_\bot)$ with $a \leq^S_\bot b$ if and only if $a = b$ or $a = \bot$, for any $a,b \in S \cup \{\bot\}$. Let $A$ and $B$ be two arbitrary sets.

1. Characterize the monotone functions $F \in \text{Mon}(A_\bot, B_\bot)$ with the property that $F(\bot) \neq \bot$;

2. Are there any monotone functions $F \in \text{Mon}(A_\bot, B_\bot)$ such that $F(a) = \bot$ for some $\bot, a \in A_\bot$?

3. For this item only, suppose that $A$ and $B$ are finite:
   (a) How many partial functions are there in $A \rightarrow B$?
   (b) How many (not necessarily monotone) total functions are there in $A_\bot \rightarrow B_\bot$?
   (c) How many monotone functions are there in $\text{Mon}(A_\bot, B_\bot)$?

4. Show that there is a bijective correspondence between partial functions in $A \rightarrow B$ and monotone functions in $\text{Mon}(A_\bot, B_\bot)$. Give an example of a total function in $A_\bot \rightarrow B_\bot$ which does not correspond to a partial function in $A \rightarrow B$.

Exercise 38. (See also Exercise 37). For any sets $A$ and $B$, $\text{Mon}(A_\bot, B_\bot) = \text{Cont}(A_\bot, B_\bot)$ and, moreover, this set of total functions bijectively corresponds to the set of partial functions $A \rightarrow B$.

Exercise 39. For any sets $A$ and $B$, $(\text{Cont}(A_\bot, B_\bot), \leq, \bot)$ and $(A \rightarrow B, \leq, \bot)$ are isomorphic bottomed CPOs.

Exercise 40. Prove that the function $F$ associated to the factorial function discussed right after Theorem 12 satisfies the hypothesis of Theorem 12 in particular that it is continuous. What if we replace “$n \ast g(n-1)$ if $n > 0$” by “$n \ast g(n-1)$ if $n > 1$”, or $n \ast g(n-1)$ by $n \ast g(n+1)$?

Exercise 41. Prove that the function $F$ associated to the context-free grammar discussed after Theorem 12 satisfies the hypothesis of Theorem 12 in particular that it is continuous. Describe a general procedure to associate such a function to any context-free grammar, so that its least fixed point is precisely the language of the grammar.

Exercise 42. Define the famous Ackermann function

$$a(m,n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  a(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  a(m-1, a(m, n-1)) & \text{if } m > 0 \text{ and } n > 0
\end{cases}$$

as a fixed-point and then represent it in the formal domain language in Figure 2.12 in two different ways: first as a (curried) function taking one argument and producing a function taking another argument and then producing an integer results, and second as a function taking a pair argument. For the latter, extend the equational theory in Figure 2.12 with a pair construct and corresponding projection operations.

2.6.7 Notes

Complete partial orders play a central role in theoretical computer science and especially in denotational semantics (Section 3.4) and in domain theory (started with the seminal paper by Dana Scott [71]). Some of the original work in these fields was done using complete lattices (i.e., partial orders in which all subsets have both a supremum and an infimum), which also admit fixed-point theorems. Nevertheless, CPOs are generally...
considered to have better properties than complete lattices, particularly in the context of high-order languages, so these days both denotational semantics and domain theory build upon CPOs instead of complete lattices.

Complete partial orders are sometimes called directed-complete partial orders, or $\omega$-complete partial orders, and abbreviated DCPOs or $\omega$-CPOs. Also, the notion of continuity as we used in this section is sometimes called Scott continuity, named after Dana Scott. There are many fixed-point theorems in mathematics and in computer science; the variant we used in this section is typically attributed to Bronisław Knaster and Alfred Tarski, and is called the Knaster-Tarski fixed-point theorem. At the time this book was being written, there was a conference totally dedicated to fixed points, called the *International Conference on Fixed Point Theory and Its Applications*. 
The construction underlying \( HO \) should be made into or at least should relate to a "model" of the equational theory in Figure 2.12. That does not seem to be easy to achieve, though. What we may be able to do instead is to define a morphism from the initial model of the equational theory into BCPO, then show that equational reasoning is sound through this morphism. This will give us the right to use ordinary rewriting to compute fixed points and hereby to execute denotational semantics.
2.7 Category Theory

Category theory appeared for at least two important reasons:

1. to capture general concepts and principles from various particular instances of mathematics; and
2. to eliminate some of the foundational concerns, especially those related to set theory.

While the usefulness of category theory is still debatable in mathematics, there is no doubt that category theory is a very powerful language to express and handle complex computer science concepts. In what follows, we shall use category theory as a means to define some very intuitive models of simply-typed \( \lambda \)-calculus. But first, let us introduce some basic notions of category theory.

**Category**

A category \( C \) consists of:

- A class of objects written \( |C| \), or \( \text{Ob}(C) \). It is called a “class” to reflect the fact that it does not need to obey the constraints of set theory; one can think of a class as something “potentially larger than a set”;
- A set of morphisms, or arrows, for any two objects \( A, B \in |C| \), written \( C(A, B) \). The fact that \( f \in C(A, B) \) is often expressed using the more familiar notation \( f : A \to B \). The object \( A \) is called the domain of \( f \), or its source, and \( B \) is called the codomain, or the target of \( f \);
- A special identity morphism \( 1_A : A \to A \) for any object \( A \in |C| \);
- A composition operator \( \vartriangleleft : C(A, B) \times C(B, C) \to C(A, C) \) for any (not necessarily distinct) objects \( A, B, C \in |C| \), with the following properties:
  - (identity) \( 1_A; f = f; 1_B = f \) for any \( A, B \in |C| \) and any \( f : A \to B \), and
  - (associativity) \( (f; g); h = (f; g); h \) for any \( f : A \to B, g : B \to C, \) and \( h : C \to D \).

Categories are everywhere. For example:

- **Set** is the category whose objects are sets and whose morphisms are the usual functions;
- **Set\text{\textsuperscript{inj}}** is the category whose objects are sets and whose morphisms are the injective functions;
- **Set\text{\textsuperscript{sur}}** is the category whose objects are sets and whose morphisms are the surjective functions.

**Exercise 43.** Prove that the above are indeed categories.

Note that it may be the case that there are no morphisms between some given objects. For example, there is no morphism in \( \text{Set}\text{\textsuperscript{inj}}([1, 2], [1]) \). Let us discuss several other categories:

- **Mon** is the category of monoids, i.e., structures \((M, \cdot_M : M \times M \to M, e \in M)\) with \( \cdot_M \) associative and identity \( e \), with structure preserving functions, i.e., functions \( f : M \to N \) such that \( f(a \cdot_M b) = f(a) \cdot_N f(b) \) and \( f(e_M) = e_N \), as morphisms;
- **Grp** is the category of groups and morphisms of groups;
- **Poset** is the category of partially ordered sets and monotone functions between them;
• **Real**≤ is the category whose objects are the real numbers and whose morphisms are given by the “≤” relation: \( p \to q \) if and only if \( p \leq q \).

**Exercise 44.** Show that the above are categories.

Let us continue the discussion on categorical concepts. Given morphisms \( f : A \to B \) and \( g : B \to A \), one can obtain the morphisms \( f;g : A \to A \) and \( g;f : B \to A \). If these morphisms are the identities on \( A \) and \( B \), respectively, the morphisms \( f \) and \( g \) are called isomorphisms and the objects \( A \) and \( B \) are said to be isomorphic. The notation \( A \cong B \) is often used to denote that \( A \) and \( B \) are isomorphic objects.

**Diagrams** A diagram in a category \( C \) is a directed graph whose nodes are objects and whose arrows are morphisms in \( C \). Formally, a diagram consists of a pair of mappings \( d : \text{Nodes} \to |C| \) and \( d : \text{Arrows} \to C \), written compactly \( d : (\text{Nodes},\text{Arrows}) \to C \), where \( (\text{Nodes},\text{Arrows}) \) is some (not necessarily finite) labeled digraph (i.e., directed, or oriented, graph), such that for any \( \alpha : i \to j \) in \( \text{Arrows} \), \( d(\alpha) \) is a morphism \( d(i) \to d(j) \) in \( C \).

To simplify writing, we draw diagrams directly as digraphs and do not specify the mapping explicitly. For example, the following nine figures represent are diagrams:

A diagram is said to commute, or is a commutative diagram, if and only if any two paths between any two nodes correspond to equal morphisms, where path concatenation is interpreted as morphism composition. For example, the top-left diagram commutes if \( f;g = h \); the top-right diagram commutes if \( f;g = u;v \); the middle diagram commutes if \( f = g \); the bottom-right diagram commutes if \( f_i;g_i = f_j;g_j \) for all numbers \( 1 \leq i, j \leq n \).

An interesting diagram in the category **Real**≤ is that of positive numbers, with an arrow \( p \to q \) if \( p \leq q \).

Unless explicitly stated differently, from now on we assume all the diagrams that we draw to be commutative.

**Cones and Limits**

96
Given a diagram $d : (\text{Nodes}, \text{Arrows}) \to C$, a cone of $d$ is a pair $(C, \{\gamma_i\}_{i \in \text{Nodes}})$, where $C$ is an object in $|C|$ and $\gamma_i : C \to d(i)$ are morphisms in $C$ with the property that $\gamma_i ; d(\alpha) = \gamma_j$ for any $\alpha : i \to j$ in $\text{Arrows}$:

\[
\begin{array}{ccc}
\gamma_i & d(i) \\
C & \downarrow d(\alpha) \\
\gamma_j & d(j)
\end{array}
\]

In other words, all the diagrams formed by the cone with any edge in the diagram commute. The terminology of “cone” probably comes from graphical resemblance with the 3D figure obtained when one regards the diagram as a disc and $C$ as a point above the plane of the disc, which is connected to every point in the diagram. But, of course, there can be all types of diagrams of all kinds of different shape.

Let us next discuss some examples of cones:

- a cone of a diagram containing just one object $A$ and no morphism is any object $C$ together with some morphism $C \to A$ (hence there is a one-to-one correspondence between morphisms of target $A$ and cones of $A$);

- a cone of a diagram containing two disconnected objects $A, B$ is any object $C$ together with morphisms $f : C \to A$, $g : C \to B$;

- a cone of an empty diagram is any object $C$ (the existence of a morphism from $C$ to any object in the diagram is vacuously fulfilled);

- a cone of a diagram consisting of just a morphism $f : A \to B$ is an object $C$ together with a morphism $g : C \to A$ (the other morphism of the cone, say $h : C \to B$, is uniquely determined as $g ; f$);

- a cone of the diagram of positive real numbers in $\text{Real}^\leq$ is uniquely determined by any negative number or zero (because these numbers are smaller than or equal to any positive real number); moreover, diagrams in $\text{Real}^\leq$ admitting cones are precisely those subsets of real numbers which have lower bounds.

A limit of a diagram $d : (\text{Nodes}, \text{Arrows}) \to C$ is a “maximal” cone $d$. Formally, a limit of $d$ is a cone $(L, \{\delta_i\}_{i \in \text{Nodes}})$ such that for any other cone $(C, \{\gamma_i\}_{i \in \text{Nodes}})$ of $d$, there is a unique morphism from $C$ to $L$, say $h : C \to L$, such that $h ; \delta_i = \gamma_i$ for all $i \in \text{Nodes}$:

\[
\begin{array}{ccc}
\gamma_i & d(i) & \delta_i \\
C & \downarrow h & \downarrow d(\alpha) \\
\gamma_j & d(j) & \delta_j
\end{array}
\]

**Exercise 45.** Any two limits of a diagram are isomorphic.

Because of this, we say that limits are taken up-to-isomorphism.

Let us next discuss some examples of limits:
• A limit of a diagram containing just one object $A$ and no morphism is any object $L$ that is isomorphic to $A$ (the isomorphism is part of the limit);

• A limit of a diagram containing two disconnected objects $A, B$ is called a \textit{product} of $A$ and $B$, and is usually written $(A \times B, \pi_A, \pi_B)$, or even more simply just $A \times B$ and the two projections $\pi_A$ and $\pi_B$ are understood - the product $A \times B$ has therefore the property that for any object $C$ and morphisms $f : C \to A$ and $g : C \to B$, there is a \textit{unique} morphism, usually written $(f, g) : C \to A \times B$, such that $(f, g)_* \pi_A = f$ and $(f, g)_* \pi_B = g$:

\[
\begin{tikzcd}
A & \times & B \\
\downarrow & \downarrow & \downarrow \\
\pi_A & \pi_B & \\
A \times B & C & \\
\end{tikzcd}
\]

• A limit of an empty diagram is called a \textit{final} object of the category $C$, usually denoted $\star$. Recall that a cone of an empty diagram was any object in $C \in \mathcal{C}$. Therefore, final objects $\star$ have the property that for any object $C \in C$ there is a unique morphism from $C$ to $\star$, usually denoted by $!_C : C \to \star$;

• A limit of a diagram consisting of just a morphism $f : A \to B$ is an object isomorphic to $A$;

• The limit of the diagram of positive real numbers in $\text{Real}^\leq$ is the number 0, together with the corresponding “less than” morphisms to any positive number. Moreover, any diagram in $\text{Real}^\leq$ consisting of a bounded set of numbers admits a limit, which is the infimum of the family; if the diagram is a (countable) decreasing sequence, then this limit is precisely the limit from mathematical analysis (this is perhaps where the name “limit” comes from).

\textbf{Products}

Products will play an important role in our subsequent developments. Therefore, we investigate them in slightly more depth here.

Note first that in particular instances of $C$, for example sets and functions, products are nothing but the usual cartesian products, consisting of pairs of elements, one in the first component and one in the second. Also, the final objects are typically one-element structures.

\textbf{Exercise 46.} \textit{Explain why in Set, the product of an empty set of sets is a one-element set.}

Given two morphisms $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$, note that there is a \textit{unique} morphism, written $f_1 \times f_2 : A_1 \times A_2 \to B_1 \times B_2$, such that the following diagram commutes:

\[
\begin{tikzcd}
A_1 & B_1 \\
\downarrow & \downarrow \\
A_1 \times A_2 & B_1 \times B_2 \\
\downarrow & \downarrow \\
A_2 & B_2 \\
\end{tikzcd}
\]

98
Exercise 47. Show that $A \times B \simeq B \times A$ for any $A, B \in |C|$.

Exercise 48. Why the morphism $f_1 \times f_2$ exists and is unique?

Exercise 49. Show that $A \simeq \star \times A$ for any $A \in |C|$.

Exercise 50. Let $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : B \rightarrow D$ be morphisms in a category with products. Show that $f ; (g, h) = (f ; g, f ; h)$.

Exercise 51. Let $f_1 : A_1 \rightarrow A_2$, $f_2 : A_2 \rightarrow A_3$ and $g_1 : B_1 \rightarrow B_2$, $g_2 : B_2 \rightarrow B_3$ be morphisms in a category with products. Show that $(f_1 ; f_2) \times (g_1 ; g_2) = (f_1 \times g_1) ; (f_2 \times g_2)$.

Exponentials

From now on we assume that our categories admit finite products, i.e., limits of finite diagrams of disconnected objects. In particular, the categories are assumed to have final objects.

Given two objects $B, C \in |C|$, an exponential of $B$ and $C$ is an object denoted $C^B$ together with a morphism $\text{app}^{B,C} : C^B \times B \rightarrow C$ such that for any $f : A \times B \rightarrow C$, there is a unique $g : A \rightarrow C^B$ such that $(g \times 1_B); \text{app}^{B,C} = f$:

$$
\begin{array}{c}
\begin{array}{ccc}
A \times B & \xrightarrow{f} & C \\
\downarrow{g \times 1_B} & & \downarrow{\text{app}^{B,C}} \\
C^B \times B & & \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{ccc}
A & \xrightarrow{g} & C^B \\
\downarrow{1_B} & & \downarrow{1_B} \\
B & & \\
\end{array}
\end{array}
$$

Proposition 13. If an exponential $C^B$ of $B$ and $C$ exists in $C$, then there is a one-to-one correspondence between the sets of morphisms $C(A \times B, C)$ and $C(A, C^B)$. The two components of this bijection, inverse to each other, are written:

$$
C(A \times B, C) \xrightarrow{\text{curry}} C(A, C^B)
$$

$$
C(A, C^B) \xrightarrow{\text{uncurry}} C(A \times B, C)
$$

Proof. Let us first define the functions $\text{curry}$ and $\text{uncurry}$. For any $f : A \times B \rightarrow C$, let $\text{curry}(f)$ be the unique morphism $g : A \rightarrow C^B$ given by the definition of the exponential, with the property that $(g \times 1_B); \text{app}^{B,C} = f$. Conversely, for any $g : A \rightarrow C^B$, let $\text{uncurry}(g)$ be the morphism $(g \times 1_B); \text{app}^{B,C} : A \times B \rightarrow C$. All we need to prove is that for any $f : A \times B \rightarrow C$ and $g : A \rightarrow C^B$, it is the case that $\text{uncurry}(\text{curry}(f)) = f$ and $\text{curry}(\text{uncurry}(g)) = g$. The first is equivalent to $(\text{curry}(f) \times 1_B); \text{app}^{B,C} = f$, which is immediate by the definition of $\text{curry}$, while the second follows by the unicity of $g$ with the property that $(g \times 1_B); \text{app}^{B,C} = f$, where $f$ is $\text{uncurry}(g)$. \qed

Exercise 52. Prove that $C(A, B) \simeq C(\star, B^A)$ whenever the exponential of $A$ and $B$ exists in $C$.

A category $C$ which admits finite products and exponentials for any two objects is called cartesian closed. For notational simplicity, a cartesian closed category is called a CCC.

Exercise 53. Show that if $A$ is an object in a CCC, then $A^* \simeq A$ and $*^A \simeq *$. 

99
Chapter 3

Executable Semantics Approaches
Dorel: make sure that the relationship to proof systems is well explained and referenced. make it clear that many of the semantics are formal proof systems, so they obey the general pattern of rules as explained in Section ??? (write this section as well in the preliminaries); explain also what a proof tree is there and then make the connection that such proof trees capture the idea of computation/derivation in the PL world. It may also be worth explaining, in the preliminaries, how general purpose proof systems can be captures as particular rewrite systems, and also give the theorem by Oliet and Meseguer stating the faithfulness of the representation.

Programming language designers and researchers need to understand the existing programming language semantical approaches well. A good understanding of semantics can help in at least two ways. On the one hand, it helps to understand how various language features can be formally defined and, implicitly, to better understand those language features. On the other hand, it helps to understand the limitations of the existing semantical approaches and, implicitly, to set up the ground for the development of novel and more powerful programming language semantical approaches.

possible and what not, as well as how it can be achieved and how it can be done.

or to

By an operational semantics of a programming language, one typically understands a collection of rules specifying how its expressions, statements, programs, etc., are evaluated or executed. These rules say how a possible implementation of a programming language should “operate” and it is not difficult in practice to derive an implementation of (an interpreter of) a language by just following and translating its operational semantics into the target implementation language.

There is no definite agreement on how an operational semantics of a language should be given, because any description of a programming language which is rigorous enough to quickly lead to a correct implementation of the language can be considered to be a valid operational semantics.

This chapter introduces and discusses several operational semantics approaches in a uniform way, by means of a common computational logic framework, namely rewrite logic. The operational semantics approaches addressed in this chapter are: natural or big-step operational semantics, small-step structural operational semantics (SOS), modular SOS, reduction semantics with evaluation contexts, and the chemical abstract machine (CHAM). The first two are the most known and widely used language definitional approaches; modular SOS and reduction semantics with evaluation contexts have been introduced to enhance the modularity and compactness of SOS, and the CHAM to address the lack of support in the other approaches for true concurrency and distributed state. Each of these language definitional styles is shown to be faithfully captured as a rewrite logic theory, in the sense that there is a one-to-one correspondence between computational steps in the original language definition and computational steps in the corresponding rewrite logic theory.

This chapter has three major goals:

1. To serve as a gentle introduction to conventional operational semantics approaches;

2. To present these various operational semantics approaches in a uniform setting, thus being able to more easily compare them to each other and understand the advantages and the disadvantages of each of them; and
3. To convey the fact that rewrite logic does not force or pre-impose any given language definitional style, and that its flexibility and ease of use makes it an appealing framework for understanding and exploring existing as well as new language definitional styles.

Each language definitional style discussed in this chapter can therefore be used as a particular definitional methodology within rewrite logic, thus enabling the language designer to use his/her favorite techniques within rewrite logic with the benefit of a unified logic and generic tool support. It is important to realize that representing a language definitional style in rewrite logic does not make that style more flexible: as it will soon become clear once we start presenting the details, the technique representing it within rewrite logic inherits the same benefits and limitations that the original definitional style had.

Even though we exemplify the techniques and proofs in this chapter with a simple language for concreteness’ sake, the process of representing each definitional style in rewrite logic and proving the faithfulness of the representation is completely general and mechanical.

---

merge this:
We will also characterize the relation between the RLS representations and their corresponding definitional style counterparts, pointing out some strengths and weaknesses for each style. The reader is referred to [34, 35, 25, 9] for further details on the described operational semantics styles.

---

following was initially included as part of the IMP language section; it should be here, perhaps rewritten

To give an operational semantics to a programming language, one typically needs a special structure, called a configuration, which comprises all the semantic ingredients needed to define how the programming language under consideration operates. For example, a configuration typically contains the program or the fragment of program that needs to be executed. In addition to code, configurations typically also contain several other pieces of information, such as the program state, memory (or heap), stacks, other resources held by each thread, etc.

In general, complex languages may require complex configurations and different languages may require different configurations. There is no absolute agreement on how to denote configurations, but it is common practice to consider them as tuples separating the various configuration components with a comma and enclosing the comma-separated list of configuration components with some special brackets, such as angle or square brackets. For example, \(c_1, c_2, \ldots, c_n\) denotes a configuration containing components \(c_1, c_2, \ldots, c_n\).

For the simple language IMP, we only need very simple configurations. In this chapter, we follow the comma-and-angle-bracket notational convention, that is, we separate the configuration components by commas and then enclose the entire list with angle brackets. For example, \(\langle a, \sigma \rangle\) is a configuration containing an arithmetic expression \(a\) and a state \(\sigma\), and \(\langle b, \sigma \rangle\) is a configuration containing a Boolean expression \(b\) and a state \(\sigma\). Some configurations may not need a state while others may not need the code. For example, \(\langle i \rangle\) is a configuration holding only the integer number \(i\) that can be obtained as a result of evaluating an arithmetic expression, while \(\langle \sigma \rangle\) is a configuration holding only one state \(\sigma\) that can be obtained after evaluating a statement. Configurations can therefore be of different types and need not necessarily have the same number of components.
It is common practice when giving “paper” (i.e., not machine executable) definitions of languages to informally and tacitly assume configurations for any types and any number of components.

An operational semantics can be viewed as a means to establish a relationship between configurations. For example, the relation defined by a small-step semantics captures exactly one desirable execution step in the defined language, while the relation defined by a big-step semantics captures several such small steps, typically relating the corresponding program or fragment of program directly to its result after evaluation.

Once one goes beyond paper definitions of languages and wants to have actual rigorous and machine executable definitions, one needs to worry about giving formal definitions of configurations. There are many possible ways to do this. Figure FIXTHIS shows an algebraic signature defining the IMP configurations needed for the subsequent operational semantics. For simplicity, we preferred to explicitly define each type of needed configuration. Consequently, our configurations definition in Figure FIXTHIS may be more verbose than an alternative polymorphic definition, but we believe that it is clearer for this simple language. We assumed that the sorts $AExp$, $BExp$, $Stmt$ and $State$ come from algebraic definitions of the IMP syntax and state, like those in Sections 3.1.1 and 3.1.2; recall that the latter adapted the algebraic definition of partial functions in Section 2.4.6 (see Figure 2.7) as explained in Section 3.1.2.

In paper definitions of languages, configurations tend to be informally and tacitly assumed available for any types and any number of components. When defining language semantics formally, configurations need to also be defined rigorously.

An SOS defines a relation on configurations, which is typically binary and thought of as a transition relation. In general, a configuration is a tuple containing a program or a fragment of program, which we may call a term over the syntax of the language, and corresponding needed semantic infrastructure, such as a state, various control stacks, etc.; however, in our simple language definition we only need configurations consisting of pairs of a term and a state.

They both consist of defining deduction (or inference, or derivation) systems for binary relations on configurations, where a configuration is typically a tuple containing some program fragment together with state infrastructure necessary to evaluate it.

Big-Step SOS, also known as natural semantics. Under big-step SOS, the atomic provable entities are relations of configurations, typically written $C \Downarrow C'$, with the meaning that $C'$ is the configuration obtained after the (complete) evaluation of $C$.

If expression evaluation is side-effect-free, then one can drop the state from the right-side configuration and thus write big-step relations as $(a,\sigma) \Downarrow (i)$;

A transition takes a configuration to another configuration (the “next step”).

An SOS defines a relation on configurations, which is typically binary and thought of as a transition relation. In general, a configuration is a tuple containing a program or a fragment of program, which we may call a term over the syntax of the language, and corresponding needed semantic infrastructure, such as a state, various control stacks, etc.; however, in our simple language definition we only need configurations consisting of pairs of a term and a state.

However, the state needs to be carried as part of the configuration that appears to the left side of each sequent, because it may be needed if expressions contain variables, to look them up.

To avoid treating special cases in the subsequent small-step definitions, we prefer to mention the state in the right-side configuration even if it does not change.

\textit{get stuff from the I&C paper}
say that can also learn these semantic definitional styles, in that we do not assume the reader necessarily familiar with those.

In this section we will investigate a particular but very common operational semantics approach, called structural operational semantics and abbreviated SOS, as well as two variations aiming at improving modularity and compactness of language definitions, namely modular SOS (MSOS) and reduction semantics with evaluation contexts. Language definitional frameworks or styles are typically called “structural” when their authors want to convey the intuition that programming languages are defined inductively over the structure of their syntax.

There are two common SOS definitional styles of programming languages, big-step SOS and small-step SOS. They both consist of defining deduction (or inference, or derivation) systems for binary relations on configurations, where a configuration is typically a tuple containing some program fragment together with state infrastructure necessary to evaluate it. Even though big-step SOS can be technically regarded as a special case of the small-step SOS where the small-step is big enough to take a program or a fragment of it directly to its final result, the two styles tend to be regarded as conceptually rather different.

• **Big-Step SOS**, also known as natural semantics. Under big-step SOS, the atomic provable entities are relations of configurations, typically written \( C \Downarrow C' \), with the meaning that \( C' \) is the configuration obtained after the (complete) evaluation of \( C \). A big-step SOS describes in a divide-and-conquer manner how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, substatements, etc.). For example, the big-step SOS definition of addition in a language whose expression evaluation can have side effects is

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1, \sigma_1 \rangle, \langle a_2, \sigma_1 \rangle \Downarrow \langle i_2, \sigma_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i, \sigma_2 \rangle}
\]

where \( i \) is the sum of \( i_1 \) and \( i_2 \) (3.1)

Here, the meaning of a relation \( \langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle \) is that arithmetic expression \( a \) is evaluated in state \( \sigma \) to integer \( i \) and new state \( \sigma' \). If expression evaluation is side-effect-free, then one can drop the state from the right-side configuration and thus write big-step relations as \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \); big-step SOS definitions transform as expected:

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle, \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i \rangle}
\]

where \( i \) is the sum of \( i_1 \) and \( i_2 \) (3.2)

• **Small-Step SOS**, also known as transitional semantics, is how SOS was originally introduced. Under small-step SOS, the atomic provable entities are one-computation-step transitions, showing how a fragment of program is advanced one step in its evaluation process; a small-step SOS identifies for each language construct typically one of its syntactic counterparts which can be advanced precisely one step, and then shows how that subcomputation step translates into a one-computation step for the language construct. For example, the small-step SOS definition of addition is

\[
\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a_2, \sigma \rangle}
\]

\[
\frac{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a'_2, \sigma \rangle}
\]

(3.3)

(3.4)
\[
\langle i_1 + i_2, \sigma \rangle \rightarrow \langle i, \sigma \rangle
\]

where \( i \) is the sum of \( i_1 \) and \( i_2 \) \hspace{1cm} (3.5)

We will be able to formally show that a step in a big-step SOS of a language corresponds to many steps in a small-step SOS of the same language.

Many recent works on SOS definitions of languages take the liberty to mix big-step and small-step. For example, one may evaluate the condition of a conditional statement in one big-step, but then transition to the left or to the right branch of the conditional in a small-step; or, more generally, one can use big-step for expressions and small-step for statements.

In our simple language definition, we only need simple configurations. We enclose the various components forming a configuration with angle brackets. For example, \( \langle a, \sigma \rangle \) is a configuration containing an arithmetic expression \( a \) and a state \( \sigma \), and \( \langle b, \sigma \rangle \) is a configuration containing a Boolean expression \( b \) and a state \( \sigma \). Configurations of different types need not necessarily have the same number of components. For example, since all programs evaluate in the initial state, there is no need to mention a state next to a program in a configuration; in this case, a configuration is simply a one element tuple, \( \langle p \rangle \), where \( p \) is a program. We will introduce configurations tacitly by need in our SOS language definitions; what distinguishes configurations from other structures are the angle brackets.

The core ingredient of an SOS definition is the sequent. Like in definitions of deductive systems in general, a sequent can be almost any tuple; however, in our particular SOS definitions, a sequent is typically a transition. We use arrows for transitions, such as \( \Downarrow \) or \( \rightarrow \). A transition takes a configuration to another configuration. For example, in a big-step SOS, a transition that signifies that arithmetic expression \( a \) evaluates in state \( \sigma \) to integer \( i \) can be written \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) if the evaluation of expressions is side-effect free, or \( \langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle \) if the evaluation of expressions may have side effects, where \( \sigma' \) is the state after the evaluation of \( a \) to \( i \) in \( \sigma \). Also, in a small-step semantics, a transition \( \langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle \) signifies that statement \( s \) in state \( \sigma \) transitions to statement \( s' \) (typically a variant of \( s \) whose computation is advanced by one step) in state \( \sigma' \).

An SOS definition is a collection of parametric rules of the form:

\[
\frac{\text{transition}_1, \text{transition}_2, \ldots, \text{transition}_k}{\text{transition}}
\]

The transitions above the line are called the condition of the rule, and the transition below the line is called the conclusion of the rule. The intuition here is that transition is possible whenever transition_1, transition_2, \ldots, transition_k are possible. We may also say that transition is derivable, or can be inferred, from transition_1, transition_2, \ldots, transition_k. This reflects the fact that an SOS definition can (and should) be viewed as a logical system, where one can deduce possible behaviors of programs.

If \( k = 0 \), then we call the rule unconditional and simply write

\[
\frac{\text{transition}}{\text{transition}}
\]

\begin{center}
\textbf{discuss } +_{\text{Int}} \text{ and other assumed builtins}
\end{center}

\begin{center}
\textbf{merge this as well:}
\end{center}

Since \( K \) makes use only of standard, context-insensitive term rewriting modulo equations, it can be executed on rewrite engines like Maude almost \textit{as is}. Nevertheless, the relationship between \( K \) and Maude is
like that between SOS, or any of the above-mentioned styles, and Maude (or ML, or Scheme, or Haskell, etc.): Maude can be used as a means to execute and analyze K language definitions. The interested reader is referred to [74] for a detailed discussion on how the various language definitional styles can be faithfully embedded in rewrite logic and then executed using systems like Maude, or even more conventional programming languages. The basic idea of these faithful embeddings is that a language definition in any of these styles, say \( L \), can be regarded as a rewrite logic theory, say \( R_L \), in such a way that there is a \textit{one-to-one computational equivalence} between reductions using \( L \) and rewrites using \( R_L \). Note that this is significantly stronger than encoding, or implementing, a framework into another framework: \( R_L \) is \( L \), not an encoding of it, the only difference between the two being insignificant notational/syntactic conventions. This is totally different from encoding \( L \) on a Turing machine or in a \( \lambda \)-calculus, for example, because such encodings would not preserve the intended computational granularity of \( L \)'s reductions (if correct, they would only preserve the relational behavior of \( L \): whatever is reduced with \( L \) can also be reduced, in any number of steps, with its encodings).

\[ \text{the essential difference between reduction semantics and rewriting semantics is that the former needs to be given permission, while the latter needs to be taken permission} \]

\[ \text{Should we consider also unravelings, or automatic elimination of conditions? Would these make them faster?} \]

See [84] for lots of citations that could go in the intro of Chapter 3. Vardejo also defines several languages in Maude, using several styles. E.g., a simple functional language in both big-step and small-step SOS, a simple imperative language in both big-step and small-step SOS, Kahn's Mini-ML using big-step SOS, Milner's CCS (he uses frozen attributes; are they really needed? I think we can use the ◦C trick to avoid it all together),

But he uses slightly different encodings/embeddings of these styles in rewrite logic, and does not state/prove general results about embeddings; he only demonstrates them by means of concrete examples. His goal is really to implement operational semantics in Maude.

\[ \text{In each figure in this section, say what the sorts of variables or the syntactic categories of metavariables are, like in the figures for big-step} \]

\[ \text{Notes} \]

- Gordon 89?
  - Centaur? Typol?

\[ \text{say that ★ means a Maude-specific section/problem/exercise} \]
Some ideas from Peter Mosses denotational semantics paper in the handbook. Most of these may go in the introduction of the book.

- syntax is about the structure of programs, whether they are legal or not.

- syntax of most languages is defined quite formally; its meaning is typically defined informally, in words; for that reason it is often the case that it is incomplete or inconsistent, unsuitable for reasoning, and useless for generation of implementations; why is the meaning of a PL not also defined formally? well, semantic frameworks are not as obvious as the syntactic ones (CFG), they have advantages and disadvantages, they are not as easy to read and understand, one needs mathematical training, etc. also, can be painful to do it, because they are non-modular (unlike the syntax).

- semantics is about their meaning, the behavior they produce when executed on computers; formal description of PLs;

- a semantics documents the design of the language; it also gives insight about the language one is defining; if executable, then a great tool for design.

- a semantics serves as a standard implementation of the language: ensures that all implementations give the same result when running the same program.

- a semantics does NOT tell how the language should be implemented! in particular, it may use additional constructs that may have no equivalent in implementations.

- a semantics gives a basis for reasoning about programs.
Int ::= the domain of (unbounded) integer numbers, with usual operations on them
Bool ::= the domain of Booleans
Id ::= standard identifiers
AExp ::= Int
       | Id
       | AExp + AExp
       | AExp / AExp
BExp ::= Bool
       | AExp <= AExp
       | ! BExp
       | BExp && BExp
Block ::= {}
       | { Stmt }
Stmt ::= Block
       | Id = AExp;
       | Stmt Stmt
       | if (BExp) Block else Block
       | while (BExp) Block
Pgm ::= int List(Id); Stmt

Figure 3.1: Syntax of IMP, a small imperative language, using algebraic BNF.

3.1 IMP: A Simple Imperative Language

To illustrate the various semantic styles discussed in this chapter, we have chosen a small imperative language, called IMP, whose syntax is inspired from C and Java. In fact, if we wrap an IMP program in a main() {...} function the we get a valid C program. IMP has arithmetic expressions which include the domain of arbitrarily large integer numbers, Boolean expressions, assignment statements, conditional statements, while loop statements, and sequential composition of statements. Statements can be grouped in blocks surrounded with curly brackets, and the branches of the conditional and the loop body are required to be blocks. All variables used in an IMP program need to be declared at the beginning of the program, can only hold integer values (for simplicity, IMP has no Boolean variables), and are initialized with default value 0.

3.1.1 IMP Syntax

We here define the syntax of IMP, first using the Backus-Naur form (BNF) notation for context-free grammars and then using the alternative and completely equivalent mixfix algebraic notation (see Section 2.1.3). The latter is in general more appropriate for semantic developments of a language.

IMP Syntax as a Context-Free Grammar

Figure 3.1 shows the syntax of IMP using the algebraic BNF notation. In this book we implicitly assume parentheses as part of any syntax, without defining them explicitly. Parentheses can be freely used for grouping, to increase clarity and/or to avoid ambiguity in parsing. For example, with the syntax in Figure 3.1 (x + 3) / y is a well-formed IMP arithmetic expression.
The only algebraic feature in the IMP syntax in Figure 3.1 is the use of \texttt{List\{ld\}} for variable declarations (last production), which in this case is clear: one can declare a comma-separated list of variables. To stay more conventional in notation, we refrained from replacing the productions \texttt{Stmt ::= \{\}} | \texttt{Stmt Stmt} with the algebraic production \texttt{Stmt ::= List\{Stmt\}} which captures the idea of statement sequentialization more naturally. Moreover, our syntax for statement sequential composition allows ambiguous parsing. Indeed, if \(s_1, s_2, s_3 \in \texttt{Stmt}\) then \(s_1 s_2 s_3\) can be parsed either as \((s_1 s_2) s_3\) or as \(s_1 (s_2 s_3)\). However, the semantics of statement sequential composition is such that the parsing ambiguity is irrelevant (but that may not always be the case). It may be worthwhile pointing out that one should not get tricked by thinking that different parsings mean different evaluation orders. In our case here, both \((s_1 s_2) s_3\) and \(s_1 (s_2 s_3)\) will proceed by evaluating the three statements in order. The difference between the two is that the former will first evaluate \(s_1 s_2\) and then \(s_3\), while the latter will first evaluate \(s_1\) and then \(s_2 s_3\); in either case, \(s_1, s_2\) and \(s_3\) will end up being evaluated in the same order: first \(s_1\), then \(s_2\), and then \(s_3\).

The IMP language constructs have their usual imperative meaning. For diversity and demonstration purposes, when giving the various semantics of IMP we will assume that + is \textit{non-deterministic} (it evaluates the two subexpressions in any order, possibly interleaving their corresponding evaluation steps), / is non-deterministic and \textit{partial} (it will stuck the program when a division by zero takes place), <= is \textit{left-right sequential} (it first evaluates the left subexpression and then the right subexpression), and that && is left-right sequential and \textit{short-circuited} (it first evaluates the left subexpression and then it conditionally evaluates the right only if the left evaluated to true).

One of the main reasons for which arithmetic language constructs like + above are allowed to be non-deterministic in language semantic definitions is because one wants to allow flexibility in how the language is implemented, and not because these operations are indeed intended to have fully non-deterministic, or random, behaviors in all implementations. In other words, their non-determinism is to a large extent an artifact of their intended \textit{underspecification}. Some language manuals actually state explicitly that one should not rely on the order in which the arguments of language constructs are evaluated. In practice, it is considered to be programmers’ responsibility to write their programs in such a way that one does not get different behaviors when the arguments are evaluated in different orders.

To better understand the existing semantic approaches and to expose some of their limitations, Section 3.5 discusses extensions of IMP with expression side effects (a variable increment operation), with abrupt termination (a halt statement), with dynamic threads and join synchronization, with local variable declarations, as well as with all of these together; the resulting language is called IMP++. The extension with side effects, in particular, makes the evaluation strategies of +, <= and && semantically relevant.

Each semantical approach relies on some basic mathematical infrastructure, such as integers, Booleans, etc., because each semantic definition reduces the semantics of the language constructs to those domains. We will assume available any needed mathematical domains, as well as basic operations on them which are clearly tagged (e.g., \(+_{int}\) for the addition of integer numbers, etc.) to distinguish them from homonymous operations which are language constructs. Unless otherwise stated, we assume no implementation-specific restrictions in our mathematical domains; for example, we assume integer numbers to be arbitrarily large rather than representable on 32 bits, etc. We can think of the underlying domains used in language semantics as parameters of the semantics; indeed, changing the meaning of these domains changes the meaning of all language semantics using them. We also assume that each mathematical domain is endowed with a special element, written \(\bot\) for all domains to avoid notational clutter, corresponding to \textit{undefined} values of that domain. Some of these mathematical domains are defined in Chapter 2, appropriate references will be given when such domains are used.
sorts:
  Int, Bool, Id, AExp, BExp, Block, Stmt, Pgm

subsorts:
  Int, Id  <  AExp
  Bool  <  BExp
  Block  <  Stmt

operations:

- _ + _ : AExp × AExp → AExp
- _ / _ : AExp × AExp → AExp
- _ <= _ : AExp × AExp → BExp
- ! _ : BExp → BExp
- _ & & _ : BExp × BExp → BExp
- {} : → Block
- {_} : Stmt → Block
- _ = _ ; : Id × AExp → Stmt
- _ _ ; : Stmt × Stmt → Stmt

if (_) . else _ : BExp × Block × Block → Stmt
while (_) _ : BExp × Block → Stmt
int _ _ ; : List {Id} × Stmt → Pgm

Figure 3.2: Syntax of IMP as an algebraic signature.

We take the freedom to tacitly use the following naming conventions for meta or mathematical variables\(^1\) ranging over IMP-specific terms throughout the remainder of this chapter: \(x, X \in Id\); \(a, A \in AExp\); \(b, B \in BExp\); \(s, S \in Stmt\); \(i, I \in Int\); \(t, T \in Bool\); \(p, P \in Pgm\). Any of these can be primed or indexed.

**IMP Syntax as an Algebraic Signature**

Following the relationship between the CFG and the mixfix algebraic notations explained in Section 2.1.3, the BNF syntax in Figure 3.1 can be associated the entirely equivalent algebraic signature in Figure 3.2 with one (mixfix) operation per production: the terminals mixed with underscores form the name of the operation and the non-terminals give its arity. This signature is easy to define in any rewrite engine or theorem prover; moreover, it can also be defined as a data-type or corresponding structure in any programming language. We next show how it can be defined in Maude.

\* Definition of IMP Syntax in Maude

Using the Maude notation for algebraic signatures, the algebraic signature in Figure 3.2 can yield the Maude syntax module in Figure 3.3. We have additionally picked some appropriate precedences and formatting attributes for the various language syntactic constructs (see Section 2.5.6 for more details on Maude and the meaning of these attributes).

\(\text{\(1\)}}\) Recall that we use an *italic* font for such variables, in contrast to the *typewriter* font that we use for code (including program variable identifiers, integers, operation symbols, etc.). For example, if we write \(x, x \in Id\) then we mean an arbitrary identifier that \(x\) refers to, and the *concrete* identifier \(x\). The latter can appear in programs, while the former cannot. The former is mainly used to define semantics or state properties of the language.
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID.

--- AExp
  sort AExp. subsorts Int Id < AExp.
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)].
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)].

--- BExp
  sort BExp. subsort Bool < BExp.
  op _<=_ : AExp AExp -> BExp [prec 37 format (d b o d)].
  op !_ : BExp -> BExp [prec 53 format (b o d)].
  op _&&_ : BExp BExp -> BExp [prec 55 gather (E e) format (d b o d)].

--- Block and Stmt
  sorts Block Stmt. subsort Block < Stmt.
  op {} : -> Block [format (b b o)].
  op {_} : Stmt -> Block [format (d n++i n--i d)].
  op _=_; : Id AExp -> Stmt [prec 40 format (d b o b o)].
  op __ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d n i) d].
  op if(_)_else_ : BExp Block Block -> Stmt [prec 59 format (b so d d s n i b o d)].
  op while(_)_ : BExp Block -> Stmt [prec 59 format (b so d d s d)].

--- Pgm
  sort Pgm.
  op int_;_ : List{Id} Stmt -> Pgm [prec 70 format (nb o d ni d)].
endm

Figure 3.3: IMP syntax as an algebraic signature in Maude. This definition assumes appropriate modules PL-INT, PL-BOOL and PL-ID defining corresponding sorts Int, Bool, and Id, respectively.

The module IMP-SYNTAX in Figure 3.3 imports three builtin modules, namely: PL-INT, which we assume it provides a sort Int; PL-BOOL, which we assume provides a sort Bool; and PL-ID which we assume provides a sort Id. We do not give the precise definitions of these modules here, particularly because one may have many different ways to define them. In our examples from here on in the rest of this chapter we assume that PL-INT contains all the integer numbers as constants of sort Int, that PL-BOOL contains the constants true and false of sort Bool, and that PL-ID contains all the letters in the alphabet as constants of sort Id. Also, we assume that the module PL-INT comes equipped with as many builtin operations on integers as needed. To avoid operator name conflicts caused by Maude’s operator overloading capabilities, we urge the reader to not use the Maude builtin INT and BOOL modules, but instead to overwrite them. Appendix A.1 shows one possible way to do this: we define new modules PL-INT and PL-BOOL “hooked” to the builtin integer and Boolean values but defining only a subset of operations on them and with clearly tagged names to avoid name overloading, e.g., _+Int_, _/Int_, etc.

Recall from Sections 2.4.6 and 2.5.6 that lists, sets, bags, and maps are trivial algebraic structures which can be easily defined in Maude; consequently, we take the freedom to use them without definition whenever needed, as we did with using the sort List{Id} in Figure 3.3.

To test the syntax, one can now parse various IMP programs, such as:

Maude> parse
  int n, s ;
  n = 100 ;
  while (!(n <= 0)) {
    s = s + n ;
    n = n + -1 ;
  }


mod IMP-PROGRAMS is including IMP-SYNTAX.
ops sumPgm collatzPgm countPrimesPgm : -> Pgm.
ops collatzStmt multiplicationStmt primalityStmt : -> Stmt.
eq sumPgm = (int n, s; n = 100; while (!(n <= 0)) { s = s + n; n = n + -1; } ).
eq collatzStmt = (while (!(n <= 1)) { s = s + 1; q = n / 2; r = q + q + 1; if (r <= n) { n = n + n + n + 1; } else { n = q; } }).
eq collatzPgm = (int m, n, q, r, s; m = 10; while (!(m <= 2)) { n = m; m = m + -1; collatzStmt } ).
eq multiplicationStmt = (--- fast multiplication (base 2) algorithm z = 0; while (!(x <= 0)) { q = x / 2; r = q + q + 1; if (r <= x) { z = z + y; } x = q; y = y + y; } ).
eq primalityStmt = (i = 2; q = n / i; t = 1; while (i <= q && 1 <= t) { x = i; y = q; multiplicationStmt if (n <= z) { t = 0; } else { i = i + 1; q = n / i; } } ).
eq countPrimesPgm = (int i, m, n, q, r, s, t, x, y, z; m = 10; n = 2; while (n <= m) { primalityStmt if (1 <= t) { s = s + 1; } else {} n = n + 1; } ).
endm

Figure 3.4: IMP programs defined in a Maude module IMP-PROGRAMS.
We may use the terminology state lookupIMPAny operational semantics of which checks whether
n
collatzStmt
The macro
variables which were not declared.

variables to integer values. Moreover, since
IMP
operations will not count as computational steps, so they will not interfere with or undesirably modify the

equations defining such map structures are computationally invisible: semantic transitions that are part of
State
and the equivalence); we let
σ
be an alias for the map sort above. From a semantic point of view, the
IMP
state operations correspond to conventional mathematical operations on
partial finite-domain functions from variables to integers in \([\text{Id} \to \text{Int}]\)\text{finite} (see Section 2.1.2) or, equivalently,
to structures of sort \(\text{Map}[\text{Id} \to \text{Int}]\) defined using equations (see Section 2.4.6 for details on the notation
and the equivalence); we let State be an alias for the map sort above. From a semantic point of view, the
equations defining such map structures are computationally invisible: semantic transitions that are part of
various IMP semantics will be performed modulo these equations. In other words, state lookup and update
operations will not count as computational steps, so they will not interfere with or undesirably modify the
intended computational granularity of the defined language.

We let \(\sigma, \sigma', \sigma_1, \text{etc.}, \) range over states. By defining IMP states as partial finite-domain functions
\(\sigma : \text{Id} \to \text{Int}, \) we have a very natural notion of undefinedness for a variable that has not been declared and thus
has not been initialized in a state: variable \(x\) is considered undefined in a state \(\sigma\) if and only if \(x \notin \text{Dom}(\sigma)\).
We may use the terminology state lookup for the operation \(\_\_\_ : \text{State} \times \text{Id} \to \text{Int}, \) the terminology state
update for the operation \(\_\_\_ \_\_ : \text{State} \times \text{Int} \times \text{Id} \to \text{State}, \) and the terminology state initialization for the operation \(\_\_\_ \_\_ \_\_. \text{List}[\text{Id}] \times \text{Int} \to \text{State}.

Recall from Section 2.1.2 that the lookup operation is itself a partial function, because the state to lookup
may be undefined in the variable of interest; as usual, we let \(\bot\) denote the undefined state and we write as
expected \(\sigma(x) = \bot\) and \(\sigma(x) \neq \bot\) when the state \(\sigma\) is undefined and, respectively, defined in variable \(x. \) Recall

Now it is a good time to define a module, say IMP-PROGRAMS, containing as many IMP programs as
one bears to write. Figure 3.4 shows such a module containing several IMP programs. Note that we took
advantage of Maude’s rewriting capabilities to save space and reuse some of the defined fragments of
programs as ‘‘macros’’. The program sumPgm calculates the sum of numbers from 1 to 100; since we do not have subtraction in IMP, we decremented the value of \(n\) by adding –1.

The program collatzPgm in Figure 3.4 tests Collatz’ conjecture for all numbers from 1 to 10, counting
the total number of steps in \(s.\) The Collatz conjecture, still unsolved, is named after Lothar Collatz (but also
known as the \(3n + 1\) conjecture), who first proposed it in 1937. Take any natural number \(n.\) If \(n\) is even, divide
it by 2 to get \(n/2;\) if \(n\) is odd multiply it by 3 and add 1 to obtain \(3n + 1.\) Repeat the process indefinitely.
The conjecture claims that no matter what number you start with, you will always eventually reach 1. Paul
Erdős said about the Collatz conjecture: “Mathematics is not yet ready for such problems.” While we do
not attempt to solve it, we can test it even in a simple language like IMP. It is a good example program to
test IMP semantics because it makes use of almost all IMP’s language constructs and also has nested loops.
The macro collatzStmt detaches the check of a single \(n\) from the top-level loop iterating \(n\) through all
\(2 < n \leq m.\) Note that, since we do not have multiplication and test for even numbers in IMP, we mimic them
using the existing IMP constructs.

Finally, the program countPrimesPgm counts all the prime numbers up to \(m.\) It uses primalityStmt,
which checks whether \(n\) is prime or not (writing \(t\) to 1 or to 0, respectively), and primalityStmt makes
use of multiplicationStmt, which implements a fast base 2 multiplication algorithm. Defining such a
module with programs helps us to test the desired language syntax (Maude will report errors if the programs
appear in the right-hand-sides of the equations are not parsable), and will also help us later on to test the
various semantics that we will define.

3.1.2 IMP State

Any operational semantics of IMP needs some appropriate notion of state, which is expected to map program
variables to integer values. Moreover, since IMP disallows uses of undeclared variables, it suffices for the
state of a given program to only map the declared variables to integer values and stay undefined in the
variables which were not declared.

Fortunately, all these desired IMP state operations correspond to conventional mathematical operations on
partial finite-domain functions from variables to integers in \([\text{Id} \to \text{Int}]\)\text{finite} (see Section 2.1.2) or, equivalently,
to structures of sort \(\text{Map}[\text{Id} \to \text{Int}]\) defined using equations (see Section 2.4.6 for details on the notation
and the equivalence); we let State be an alias for the map sort above. From a semantic point of view, the
equations defining such map structures are computationally invisible: semantic transitions that are part of
various IMP semantics will be performed modulo these equations. In other words, state lookup and update
operations will not count as computational steps, so they will not interfere with or undesirably modify the
intended computational granularity of the defined language.

We let \(\sigma, \sigma', \sigma_1, \text{etc.}, \) range over states. By defining IMP states as partial finite-domain functions
\(\sigma : \text{Id} \to \text{Int}, \) we have a very natural notion of undefinedness for a variable that has not been declared and thus
has not been initialized in a state: variable \(x\) is considered undefined in a state \(\sigma\) if and only if \(x \notin \text{Dom}(\sigma)\).
We may use the terminology state lookup for the operation \(\_\_\_ : \text{State} \times \text{Id} \to \text{Int}, \) the terminology state
update for the operation \(\_\_\_ \_\_ : \text{State} \times \text{Int} \times \text{Id} \to \text{State}, \) and the terminology state initialization for the operation \(\_\_\_ \_\_ \_\_. \text{List}[\text{Id}] \times \text{Int} \to \text{State}.

Recall from Section 2.1.2 that the lookup operation is itself a partial function, because the state to lookup
may be undefined in the variable of interest; as usual, we let \(\bot\) denote the undefined state and we write as
expected \(\sigma(x) = \bot\) and \(\sigma(x) \neq \bot\) when the state \(\sigma\) is undefined and, respectively, defined in variable \(x. \) Recall
mod STATE is including PL-INT + PL-ID.
sort State.

op _|->_ : List{Id} Int -> State [prec 0].
op .State : -> State.
op _&_ : State State -> State [assoc comm id: .State format(d s s d)].

op _(_) : State Id -> Int [prec 0]. --- lookup
op _[_/_] : State Int Id -> State [prec 0]. --- update

var Sigma : State. var I I' : Int. var X X' : Id. var XL : List{Id}.

eq X |-> undefined = .State . --- "define" a variable in a state

eq (Sigma & X |-> I)(X) = I .
eq Sigma(X) = undefined [owise] .
eq Sigma[I / X] = (Sigma & X |-> I) [owise] .
eq (X,X',XL) |-> I = X |-> I & X' |-> I & XL |-> I .
eq .List{Id} |-> I = .State .
endm

Figure 3.5: The IMP state defined in Maude.

also from Section 2.1.2 that the update operation can be used not only to update maps but also to “define” particular elements in their domain: σ[l/x] is the same as σ in all elements different from x and is undefined in x. Finally, recall also from Section 2.1.2 that the initialization operation yields a partial function mapping each element in the first list argument to the element given as second argument. These can be easily defined equationally, following the equational approach to partial finite-domain functions in Section 2.4.6.

Definition of IMP State in Maude

Figure 3.5 adapts the generic Maude definition of partial finite-domain functions in Section 2.5.6 for our purpose here: the generic sorts Source and Target are replaced by Id and Int, respectively. Recall from Section 2.5.6 that the constant undefined has sort Undefined, which is a sub-sort of all sorts corresponding to mathematical domains (e.g., Int, Bool, etc.). This way, identifiers can be made “undefined” in a state by simply updating them with undefined (see the equation dissolving undefined bindings in Figure 3.5).

To avoid overloading the comma “,” construct for too many purposes (which particularly may confuse Maude’s parser), we took the freedom to rename the associative and commutative construct for states to &. The only reason for which we bother to give this obvious module here is because we want the various subsequent semantics of the IMP language, all of them including the module STATE in Figure 3.5, to be self-contained and executable in Maude by simply executing all the Maude code in the figures in this chapter.

3.1.3 Notes

The style that we follow in this chapter, namely to pick a simple language and then demonstrate the various language definitional approaches by means of that simple language, is quite common. In fact, we named our language IMP after a similar language introduced by Winskel in his book [87], also called IMP, which
is essentially identical to ours except that it uses a slightly different syntax and does not have variable declarations. For example, Winskel’s IMP uses “:=” for assignment and “;” as statement separator instead of statement terminator, while our IMP’s syntax resembles that of common languages like C and Java. Also, since most imperative languages do have variable declarations, we feel it is instructive to include them in our simple language. Winskel gives his IMP a big-step SOS, a small-step SOS, a denotational semantics, and an axiomatic semantics. Later, Nipkow [55] formalized all these semantics of IMP in the Isabelle/HOL proof assistant [56], and used it to formally relate the various semantics, effectively mechanizing most of Winskel’s paper proofs; in doing so, Nipkow [55] found several minor errors in Winskel’s proofs, thus showing the benefits of mechanization.

Vardejo and Martí-Oliet [83, 84] show how to use Maude to implement executable semantics for several languages following both big-step and small-step SOS approaches. Like us, they also demonstrate how to define different semantics for the same simple language using different styles; they do so both for an imperative language (very similar to our IMP) and for a functional language. Serbanuta et al. [74] use a similar simple imperative language to also demonstrate how to use rewrite logic to define executable semantics. In fact, this chapter is an extension of [74], both in breadth and in depth. For example, we state and prove general faithful rewrite logic representation results for each of the semantic approaches, while [74] did the same only for the particular simple imperative language considered there. Also, we cover new approaches here, such as denotational semantics, which were not covered in [83, 84, 74].
3.2 Big-Step Structural Operational Semantics (Big-Step SOS)

Known also under the names natural semantics, relational semantics, and evaluation semantics, big-step structural operational semantics, or big-step SOS for short, is the most “denotational” of the operational semantics: one can view big-step definitions as definitions of functions, or more generally of relations, interpreting each language construct in an appropriate domain. Big-step semantics is so easy and natural to use, that one is strongly encouraged to use it whenever possible. Unfortunately, as discussed in Section 3.10, big-step semantics has a series of limitations making it inconvenient or impossible to use in many situations, such as when defining control-intensive language features, or non-deterministic ones, or concurrency.

A big-step SOS of a programming language or calculus is given as a formal proof system (see Section 2.1.5). The big-step SOS sequents are relations over configurations, typically written \( C \Rightarrow R \) or \( C \Downarrow R \), with the meaning that \( R \) is the configuration obtained after the (complete) evaluation of \( C \). In this book we prefer the notation \( C \Downarrow R \). A big-step SOS rule therefore has the form

\[
\frac{C_1 \Downarrow R_1 \quad C_2 \Downarrow R_2 \quad \ldots \quad C_n \Downarrow R_n}{C_0 \Downarrow R_0} \quad \text{[if condition]}
\]

where \( C_0, C_1, C_2, \ldots, C_n \) are configurations holding fragments of program together with all the needed semantic components, where \( R_0, R_1, R_2, \ldots, R_n \) are result configurations, or irreducible configurations, i.e., configurations which cannot be reduced anymore, and where condition is an optional side condition; as discussed in Section 2.1.5 the role of side conditions is to filter out undesirable instances of the rule.

A big-step semantics compositionally describes how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, etc.). For example, the big-step semantics of IMP’s addition is

\[
\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle \\
\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i_1 + i_2 \rangle
\]

Here, the meaning of a relation \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) is that arithmetic expression \( a \) is evaluated in state \( \sigma \) to integer \( i \). If expression evaluation has side-effects, then one has to also include a state in the right configurations, so they become of the form \( \langle i, \sigma \rangle \) instead of \( \langle i \rangle \), as discussed in Section 3.10.

It is common in big-step semantics to not wrap single values in configurations, that is, to write \( \langle a, \sigma \rangle \Downarrow i \) instead of \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) and similarly for all the other sequents. Also, while the angle-bracket-and-comma notation \( \langle \text{code}, \text{state}, \ldots \rangle \) is common for configurations, it is not enforced; some prefer to use a square or curly bracket notation of the form [code, state, . . . ] or {code, state, . . . }, or the simple tuple notation (code, state, . . . ), or even to use a different (from comma) symbol to separate the various configuration ingredients, e.g., \( \langle \text{code} \mid \text{state} \mid \ldots \rangle \), etc. Moreover, we may even encounter in the literature sequent notations of the form \( \sigma + a \Rightarrow i \) instead of \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \), as well as variants of sequent notations that prefer to move various semantic components from the configurations into special, sometimes rather informal, decorations of the symbols \( \Downarrow \), \(-\), and/or \( \Rightarrow \).

For the sake of a uniform notation, in particular when transitioning from languages whose expressions have no side effects to languages whose expressions do have side effects (as we do in Section 3.10), we prefer to always write big-step sequents as \( C \Downarrow R \), and always use the angle brackets to surround both configurations involved. This solution is the most general; for example, any additional semantic data or labels that one may need in a big-step definition can be uniformly included as additional components in the configurations (the left ones, or the right ones, or both).
sorts:
  Configuration

operations:
  \( \langle \_ \_ \rangle : \text{AExp} \times \text{State} \rightarrow \text{Configuration} \)
  \( \langle \_ \rangle : \text{Int} \rightarrow \text{Configuration} \)
  \( \langle \_ \_ \rangle : \text{BExp} \times \text{State} \rightarrow \text{Configuration} \)
  \( \langle \_ \rangle : \text{Bool} \rightarrow \text{Configuration} \)
  \( \langle \_ \_ \rangle : \text{Stmt} \times \text{State} \rightarrow \text{Configuration} \)
  \( \langle \_ \rangle : \text{State} \rightarrow \text{Configuration} \)
  \( \langle \_ \_ \rangle : \text{Pgm} \rightarrow \text{Configuration} \)

Figure 3.6: IMP big-step configurations as an algebraic signature.

3.2.1 IMP Configurations for Big-Step SOS

For the big-step semantics of the simple language IMP, we only need very simple configurations. We follow the comma-and-angle-bracket notational convention, that is, we separate the configuration components by commas and then enclose the entire list with angle brackets. For example, \( \langle a, \sigma \rangle \) is a configuration containing an arithmetic expression \( a \) and a state \( \sigma \), and \( \langle b, \sigma \rangle \) is a configuration containing a Boolean expression \( b \) and a state \( \sigma \). Some configurations may not need a state while others may not need the code. For example, \( \langle i \rangle \) is a configuration holding only the integer number \( i \) that can be obtained as a result of evaluating an arithmetic expression, while \( \langle \sigma \rangle \) is a configuration holding only one state \( \sigma \) that can be obtained after evaluating a statement. Configurations can therefore be of different types and need not necessarily have the same number of components. Here are all the configuration types needed for the big-step semantics of IMP:

- \( \langle a, \sigma \rangle \) grouping arithmetic expressions \( a \) and states \( \sigma \);
- \( \langle i \rangle \) holding integers \( i \);
- \( \langle b, \sigma \rangle \) grouping Boolean expressions \( b \) and states \( \sigma \);
- \( \langle t \rangle \) holding truth values \( t \in \{\text{true}, \text{false}\} \);
- \( \langle s, \sigma \rangle \) grouping statements \( s \) and states \( \sigma \);
- \( \langle \sigma \rangle \) holding states \( \sigma \);
- \( \langle p \rangle \) holding programs \( p \).

IMP Big-Step SOS Configurations as an Algebraic Signature

The configurations above were defined rather informally as tuples of syntax and/or states. There are many ways to rigorously formalize them, all building upon some formal definition of state (besides IMP syntax). Since we have already defined states as partial finite-domain functions (Section 3.1.2) and have already shown how partial finite-domain functions can be formalized as algebraic specifications (Section 2.4.6), we also formalize configurations algebraically.

Figure 3.6 shows an algebraic signature defining the IMP configurations needed for the subsequent big-step operational semantics. For simplicity, we preferred to explicitly define each type of needed configuration.
Consequently, our configurations definition in Figure 3.6 may be more verbose than an alternative polymorphic definition, but we believe that it is clearer for this simple language. We assumed that the sorts \( AExp, BExp, Stmt, Pgm \) and \( State \) come from algebraic definitions of the IMP syntax and state, like those in Sections 3.1.1 and 3.1.2; recall that the latter adapted the algebraic definition of partial functions in Section 2.4.6 (see Figure 2.7) as explained in Section 3.1.2.

### 3.2.2 The Big-Step SOS Rules of IMP

Figure 3.7 shows all the rules in our IMP big-step operational semantics proof system. Recall that the role of a proof system is to derive sequents, or facts. The facts that our proof system will derive have the forms \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \), \( \langle b, \sigma \rangle \Downarrow \langle t \rangle \), \( \langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle \), and \( \langle p \rangle \Downarrow \langle \sigma \rangle \) where \( a \) ranges over \( AExp \), \( b \) over \( BExp \), \( s \) over \( Stmt \), \( p \) over \( Pgm \), \( i \) over \( Int \), \( t \) over \( Bool \), and \( \sigma \) and \( \sigma' \) over \( State \).

Informally\(^2\), the meaning of derived triples of the form \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) is that the arithmetic expression \( a \) evaluates/executes/transition to the integer \( i \) in state \( \sigma \); the meaning of \( \langle b, \sigma \rangle \Downarrow \langle t \rangle \) is similar but with Boolean values instead of integers. The reason for which it suffices to derive such simple facts is because the evaluation of expressions in our simple IMP language is side-effect-free. When we add the increment operation \( ++x \) in Section 3.10, we will have to change the big-step semantics to work with 4-tuples of the form \( \langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle \) and \( \langle b, \sigma \rangle \Downarrow \langle t, \sigma' \rangle \) instead. The meaning of \( \langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle \) is that the statement \( s \) takes state \( \sigma \) to state \( \sigma' \). Finally, the meaning of pairs \( \langle p \rangle \Downarrow \langle \sigma \rangle \) is that the program \( p \) yields state \( \sigma \) when executed in the initial state.

In the case of our simple IMP language, the transition relation is going to be deterministic, in the sense that \( i_1 = i_2 \) whenever \( \langle a, \sigma \rangle \Downarrow \langle i_1 \rangle \) and \( \langle a, \sigma \rangle \Downarrow \langle i_2 \rangle \) can be deduced (and similarly for Boolean expressions, statements, and programs). However, in the context of non-deterministic languages, triples \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) state that \( a \) may evaluate to \( i \) in state \( \sigma \), but it may also evaluate to other integers (and similarly for Boolean expressions, statements, and programs).

The proof system in Figure 3.7 contains one or two rules for each language construct, capturing its intended evaluation relation. Recall from Section 2.1.3 that proof rules are in fact rule schemas, that is, they correspond to (recursively enumerable) sets of rule instances, one for each concrete instance of the rule parameters (i.e., \( a, b, \sigma, \sigma' \), etc.). We next discuss each of the rules in Figure 3.7.

The rules (\textsc{BigStep-Int}) and (\textsc{BigStep-Lookup}) define the obvious semantics of integers and of variable lookup; these rules have no premises because integers and variables are atomic expressions, so one does not need to evaluate any other subexpression in order to evaluate them. The rule (\textsc{BigStep-Add}) has already been discussed at the beginning of Section 3.2, and (\textsc{BigStep-Div}) is similar. Note that the rules (\textsc{BigStep-Lookup}) and (\textsc{BigStep-Div}) have side conditions. We chose not to short-circuit the division operation when \( a_1 \) evaluates to 0. Consequently, no matter whether \( a_1 \) evaluates to 0 or not, \( a_2 \) is still expected to produce a correct value in order for the rule (\textsc{BigStep-Div}) to be applicable (e.g., \( a_2 \) cannot perform a division by 0).

Before we continue with the remaining rules, let us clarify, using concrete examples, what it means for rule schemas to admit multiple instances and how these can be used to derive proofs. For example, a possible instance of rule (\textsc{BigStep-Div}) can be the following (assume that \( x, y \in Id \)):

\[
\langle x, (x \mapsto 8, y \mapsto 0) \rangle \Downarrow \langle 8 \rangle \quad \langle 2, (x \mapsto 8, y \mapsto 0) \rangle \Downarrow \langle 2 \rangle \\
\langle x / 2, (x \mapsto 8, y \mapsto 0) \rangle \Downarrow \langle 4 \rangle
\]

\(^2\)Formal definitions of these concepts can only be given after one has a formal language definition. We formally define the notions of evaluation and termination in the context of the IMP language in Definition 20.

119
Figure 3.7: **BigStep**(IMP) — Big-step SOS of IMP ($i, i_1, i_2 \in \text{Int}; x \in \text{Id}; xl \in \text{List(Id)}; a, a_1, a_2 \in \text{AExp}; t \in \text{Bool}; b, b_1, b_2 \in \text{BExp}; s, s_1, s_2 \in \text{Stmt}; \sigma, \sigma', \sigma_1, \sigma_2 \in \text{State}$).
Another instance of rule (BigStep-Div) is the following, which, of course, seems problematic:

$$
\frac{⟨x, (x → 8, y → 0)⟩ ↓ 8 \quad ⟨2, (x → 8, y → 0)⟩ ↓ 4}{⟨x / 2, (x → 8, y → 0)⟩ ↓ 2}
$$

The rule above is indeed a correct instance of (BigStep-Div), but, however, one will never be able to infer $⟨2, (x → 8, y → 0)⟩ ↓ 4$, so this rule can never be applied in a correct inference.

Note, however, that the following is not an instance of (BigStep-Div), no matter what $?$ is chosen to be ($⊥$, or $8/_{\text{Int}} 0$, etc.):

$$
\frac{⟨x, (x → 8, y → 0)⟩ ↓ 8 \quad ⟨y, (x → 8, y → 0)⟩ ↓ 0}{⟨x / y, (x → 8, y → 0)⟩ ↓ (?)}
$$

Indeed, the above does not satisfy the side condition of (BigStep-Div).

The following is a valid proof derivation, where $x, y \in Id$ and $σ ∈ State$ with $σ(x) = 8$ and $σ(y) = 0$:

$$
\frac{\cdot \quad \cdot}{\langle x, σ⟩ ↓ ⟨8⟩ \quad ⟨y / x, σ⟩ ↓ ⟨0⟩ \quad ⟨2, σ⟩ ↓ ⟨2⟩ \quad ⟨y / x + 2, σ⟩ ↓ ⟨2⟩}
$$

$$
\frac{\cdot}{\langle x / (y / x + 2), σ⟩ ↓ ⟨4⟩}
$$

The proof above can be regarded as a tree, with dots as leaves and instances of rule schemas as nodes. We call such complete (in the sense that their leaves are all dots and their nodes are correct rule instances) trees proof trees. This way, we have a way to mathematically derive facts, or sequents, about programs directly within their semantics. We may call the root of a proof tree the fact (or sequent) that was proved or derived, and the tree its proof or derivation.

Recall that our original intention was, for demonstration purposes, to attach various evaluation strategies to the arithmetic operations. We wanted $+$ and $/$ to be non-deterministic and $\leq$ to be left-right sequential; a non-deterministic evaluation strategy means that the subexpressions are evaluated in any order, possibly interleaving their evaluation steps, which is different from non-deterministically picking an order and then evaluating the subexpressions sequentially in that order. As an analogy, the former corresponds to evaluating the subexpressions concurrently on a multi-threaded machine, while the latter to non-deterministically queuing the subexpressions and then evaluating them one by one on a sequential machine. The former has obviously potentially many more possible behaviors than the latter. Note that many programming languages opt for non-deterministic evaluation strategies for their expression constructs precisely to allow compilers to evaluate them in any order or even concurrently; some language manuals explicitly warn the reader not to rely on any evaluation strategy of arithmetic constructs when writing programs.

Unfortunately, big-step semantics is not appropriate for defining non-deterministic evaluation strategies, because such strategies are, by their nature, small-step. One way to attempt to do it is to work with sets of result configurations instead of just with result configurations and thus associate to each fragment of program in a state the set of all the results that it can non-deterministically yield. However, such an approach would significantly complicate the big-step definition, so we prefer to not do it. Moreover, since IMP has no side effects yet (we will add it side effects in Section 3.5), the non-deterministic evaluation strategies would not lead to non-deterministic results anyway.

We next discuss the big-step rules for Boolean expressions. The rule (BigStep-Bool) is similar to rule (BigStep-Int), but it has only two instances, one for $t = \text{true}$ and one for $t = \text{false}$. The rule (BigStep-Leq) allows to derive Boolean sequents from arithmetic ones; although we want $\leq$ to evaluate
its arguments from left to right, there is no need to do anything more at this stage, because expressions have no side effects in IMP; in Section 3.5 we will see how to achieve the desired evaluation strategy in the presence of side effects in expressions. The rules (BigStep-Not-True) and (BigStep-Not-False) are clear; they could have been combined into only one rule if we had assumed our builtin \( \text{Bool} \) equipped with a negation operation. Unlike the division, the conjunction has a short-circuited semantics: if the first conjunct evaluates to \text{false} then the entire conjunction evaluates to \text{false} (rule (BigStep-And-False)), and if the first conjunct evaluates to \text{true} then the conjunction evaluates to whatever truth value the second conjunct evaluates (rule (BigStep-And-True)).

The role of statements in a language is to change the program state. Consequently, the rules for statements derive triples of the form \( \langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle \) with the meaning that if statement \( s \) is executed in state \( \sigma \) and \text{terminates}, then the resulting state is \( \sigma' \). We will shortly discuss the aspect of termination in more detail. Rule (BigStep-Empty-Block) states that \( \emptyset \) does nothing with the state. (BigStep-Block) states that the block has the same semantics as the enclosed statement, which is correct at this stage because IMP has no local variable declarations; this will change in Section 3.5 (BigStep-Asgn) shows how the state \( \sigma \) gets updated by an assignment statement “\( x = a \)”; after \( a \) is evaluated in state \( \sigma \) using the rules for arithmetic expressions discussed above. (BigStep-Seq) shows how the state updates are propagated by the sequential composition of statements, and rules (BigStep-If-True) and (BigStep-If-False) show how the conditional first evaluates its condition and then, depending upon the truth value of that, it either evaluates its then-branch or its else-branch, but never both. The rules giving the big-step semantics of the while loop say that if the condition evaluates to \text{false} then the while loop dissolves and the state stays unchanged, and if the condition evaluates to \text{true} then the body followed by the very same while loop is evaluated (rule (BigStep-While-True)). Finally, (BigStep-Pgm) gives the semantics of programs as the semantics of their statement in a state instantiating all the declared variables to \( 0 \).

**On Proof Derivations, Evaluation, and Termination**

So far we have used the words “evaluation” and “termination” informally. In fact, without a formal definition of a programming language, there is no other way, but informal, to define these notions. Once one has a formal definition of a language, one can not only formally define important concepts like evaluation and termination, but can also rigorously reason about programs.

**Definition 20.** Given appropriate IMP configurations \( C \) and \( R \), the IMP big-step sequent \( C \Downarrow R \) is derivable, written BigStep(IMP) \( \vdash C \Downarrow R \) iff there is some proof tree rooted in \( C \Downarrow R \) which is derivable using the proof system BigStep(IMP) in Figure 3.7. Arithmetic (resp. Boolean) expression \( a \in AExp \) (resp. \( b \in BExp \)) evaluates to integer \( i \in \text{Int} \) (resp. to truth value \( t \in \{ \text{true}, \text{false} \} \) in state \( \sigma \in \text{State} \) iff BigStep(IMP) \( \vdash \langle a, \sigma \rangle \Downarrow \langle i \rangle \) (resp. BigStep(IMP) \( \vdash \langle b, \sigma \rangle \Downarrow \langle t \rangle \)). Statement \( s \) terminates in state \( \sigma \) iff BigStep(IMP) \( \vdash \langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle \) for some \( \sigma' \in \text{State} \); if that is the case, then we say that \( s \) evaluates in state \( \sigma \) to state \( \sigma' \), or that it takes state \( \sigma \) to state \( \sigma' \). Finally, program \( p \) terminates iff BigStep(IMP) \( \vdash \langle p \rangle \Downarrow \langle \sigma \rangle \) for some \( \sigma \in \text{State} \).

There are two reasons for which an IMP statement \( s \) may not terminate in a state \( \sigma \): because it may contain a loop that does not terminate, or because it performs a division by zero and thus the rule (BigStep-Div) cannot apply. In the former case, the process of proof search does not terminate, while in the second case the process of proof search terminates in principle, but with a failure to find a proof. Unfortunately, big-step semantics cannot make any distinction between the two reasons for which a proof derivation cannot be found. Hence, the termination notion in Definition 20 rather means termination with no error. To catch
division-by-zero within the semantics, we need to add a special error value that a division by zero would evaluate to, and then to propagate it through all the language constructs (see Exercise 56).

A formal definition of a language allows to also formally define what it means for the language to be deterministic and to also prove it. For example, we can prove that if an IMP program \( p \) terminates then there is a unique state \( \sigma \) such that \( \text{BigStep}(\text{IMP}) \vdash (p) \Downarrow (\sigma) \) (see Exercise 57).

Since each rule schema comprises a recursively enumerable collection of concrete instances and since we have a finite set of rule schemata, we can enumerate all the instances of all these rules. Furthermore, since proof trees built with nodes in a recursively enumerable set are themselves recursively enumerable, it follows that the set of proof trees derivable with the proof system in Figure 3.7 is recursively enumerable. In other words, we can find an algorithm that enumerates all the proof trees, in particular one that enumerates all the derivable sequents \( C \Downarrow R \). By enumerating all proof trees, given a terminating IMP program \( p \), one can eventually find the unique state \( \sigma \) such that \( (p) \Downarrow (\sigma) \) is derivable. This simple-minded algorithm may take a very long time and a huge amount of resources, but it is insightful to understand that it can be done.

It can be shown that there is no algorithm, based on proof derivation like above or on anything else, which takes as input an IMP program and says whether it terminates or not (see Exercise 58). This follows from the fact that our simple language, due to its while loops and arbitrarily large integers, is Turing-complete. Thus, if one were able to decide termination of programs in our language then one would also be able to decide termination of Turing machines, contradicting one of the basic undecidable problems, the halting problem (see Section 2.2.1 for more on Turing machines).

An interesting observation here is that non-termination of a program corresponds to lack of proof, and that the latter is not decidable in many interesting logics. Indeed, in complete logics, that is, logics that admit a complete proof system, one can enumerate all the truths. However, in general there is not much one can do about non-truths, because the enumeration algorithm will loop forever when run on a non-truth. In decidable logics one can enumerate both truths and non-truths; clearly, decidable logics are not powerful enough for our task of defining programming languages, precisely because of the halting problem argument above.

### 3.2.3 Big-Step SOS in Rewrite Logic

Due to its straightforward recursive nature, big-step semantics is typically easy to represent in other formalisms and also easy to translate into interpreters for the defined languages in any programming language. (The difficulty with big-step semantics is to actually give big-step semantics to complex constructs, as illustrated and discussed in Section 3.5.5) It should therefore come at no surprise to the reader that one can associate a conditional rewrite rule to each big-step rule and hereby obtain a rewrite logic theory that faithfully captures the big-step definition.

In this section we first show that any big-step operational semantics \( \text{BigStep} \) can be mechanically translated into a rewrite logic theory \( \mathcal{R}_{\text{BigStep}} \) in such a way that the corresponding derivation relations are step-for-step equivalent, that is, \( \text{BigStep} \vdash C \Downarrow R \) if and only if \( \mathcal{R}_{\text{BigStep}} \vdash \mathcal{R}_{C \Downarrow R} \), where \( \mathcal{R}_{C \Downarrow R} \) is the corresponding syntactic translation of the big-step sequent \( C \Downarrow R \) into a (one-step) rewrite rule. Second, we apply our generic translation technique on the big-step operational semantics \( \text{BigStep}(\text{IMP}) \) and obtain a rewrite logic semantics of IMP that is step-for-step equivalent to the original big-step semantics of IMP. Finally, we show how \( \mathcal{R}_{\text{BigStep}(\text{IMP})} \) can be seamlessly defined in Maude, thus yielding an interpreter for IMP essentially for free.
Faithful Embedding of Big-Step SOS into Rewrite Logic

To define our translation generically, we need to make some assumptions about the existence of an algebraic axiomatization of configurations. More precisely, as also explained in Section 2.1.3 we assume that for any parametric term \( t \) (which can be a configuration, a condition, etc.), the term \( \bar{t} \) is an equivalent algebraic variant of \( t \) of appropriate sort. For example, a parametric configuration \( C \) is a configuration that may possibly make use of parameters, such as \( a \in AExp, \sigma \in State, \) etc.; by equivalent algebraic variant we mean a term \( \bar{C} \) of sort Configuration over an appropriate signature of configurations like the one that we defined for IMP in Section 3.2.1 (see Figure 3.6); moreover, each parameter in \( C \) gets replaced by a variable of corresponding sort in \( \bar{C} \). Similarly, the algebraic variant of a rule side condition is an appropriate term of sort Bool.

To have a formal mechanism for performing reasoning within the employed mathematical domains, which is tacitly assumed in the big-step semantics (e.g., \( 3 + 5 = 8 \), etc.), in particular for formally evaluating side conditions, we assume that the algebraic signature associated to the various syntactic and semantic categories is extended into a background algebraic specification capable of proving precisely all the equalities over ground terms (i.e., terms containing no variables). As discussed in Section 2.4, this assumption is quite reasonable, because any computational domain is isomorphic to an initial algebra over a finite algebraic specification, and that the latter can prove precisely all the ground equational properties of the domain/initial algebra. Consider, for example, the side condition \( \sigma(x) \neq \bot \) of the rules (BigStep-Lookup) and (BigStep-Asn) in Figure 3.7. Its algebraic variant is the term \( \sigma(X) \neq \bot \) of Bool sort, where \( \sigma \) and \( X \) are variables of sorts State and Id, respectively. We therefore assume that any ground instance of this Bool term (obtained for a ground/concrete instance of the variables \( \sigma \) and \( X \)) can be proved using the background algebraic theory equal to either true (which means that the map is defined in the given variable) or false.

Consider now a general-purpose big-step rule of the form

\[
\frac{C_1 \Downarrow R_1 \quad C_2 \Downarrow R_2 \quad \ldots \quad C_n \Downarrow R_n}{C_0 \Downarrow R_0} \quad \text{[if condition]}
\]

where \( C_0, C_1, C_2, \ldots, C_n \) are configurations holding fragments of program together with all the needed semantic components, \( R_0, R_1, R_2, \ldots, R_n \) are result configurations, and condition is some optional side condition. As one may expect, we translate it into the rewrite logic rule

\[
(\forall X) \bar{C_0} \rightarrow \bar{R_0} \quad \text{if} \quad \bar{C_1} \rightarrow \bar{R_1} \wedge \bar{C_2} \rightarrow \bar{R_2} \wedge \ldots \wedge \bar{C_n} \rightarrow \bar{R_n} \wedge \text{condition}.
\]

where \( X \) is the set of parameters, or meta-variables, that occur in the big-step proof rule (schema), now regarded as variables. Therefore, the big-step SOS rule premises and side conditions are both turned into conditions of the corresponding rewrite rule. The sequent premises become rewrites in the condition, while the side conditions become simple Boolean checks.

We make two reasonable assumptions about big-step semantics: (1) configurations cannot be nested; and (2) result configurations are irreducible.

**Theorem 13. (Faithful embedding of big-step SOS into rewrite logic)** For any big-step operational semantics definition BigStep, and any BigStep appropriate configuration \( C \) and result configuration \( R \), the following equivalence holds

\[
\text{BigStep} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \bar{C} \xrightarrow{1} \bar{R},
\]

where \( \mathcal{R}_{\text{BigStep}} \) is the rewrite logic semantic definition obtained from BigStep by translating each rule in BigStep as above. (Recall from Section 2.5 that \( \xrightarrow{1} \) is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewrite logic. Also, since \( C \) and \( R \) are parameter-free—parameters only appear in rules—, \( \bar{C} \) and \( \bar{R} \) are ground terms.)
Proof. Before we proceed with the proof, let us understand how the assumptions about BigStep and the use of $\rightarrow_1$ affect the rewrite logic proof derivations that can be performed with $R_{\text{BigStep}}$. First, note that the Reflexivity and Transitivity proof rules of rewrite logic (see Section 2.5) will never apply. Second, since the rules in $R_{\text{BigStep}}$ correspond to rules in BigStep between configurations and there are no other rules in $R_{\text{BigStep}}$, and since configurations cannot be nested, we conclude that the Congruence rule of rewrite logic will never apply either. Third, since the rules of BigStep add no equations to $R_{\text{BigStep}}$, the equations of $R_{\text{BigStep}}$ correspond all to the background algebraic theory used for domain reasoning; thus, the Equality rule of rewrite logic can only instantiate rules in $R_{\text{BigStep}}$ by substituting their variables with terms (which can be ground or not), but it can perform no “inner” concurrent rewrites. Therefore, the capabilities of rewrite logic are significantly crippled by $R_{\text{BigStep}}$, as the only deductions that $R_{\text{BigStep}}$ can perform are domain (equational) reasoning and rule instantiation.

Let $\Sigma$ be the signature of the assumed background algebraic formalization of configurations.

Let us first assume that $\text{BigStep} \vdash C \Downarrow R$ and let us prove that $R_{\text{BigStep}} \vdash \overline{C} \rightarrow_1 \overline{R}$. We do this proof by structural induction on the BigStep proof/derivation tree of $C \Downarrow R$. The last proof step in the tree deriving $C \Downarrow R$ must correspond to some instance of a rule (schema) of the form

\[
\frac{C_1 \Downarrow R_1 \ldots C_m \Downarrow R_m}{C_0 \Downarrow R_0} \quad \text{if condition}
\]

where $m \geq 0$ and where the condition may be missing, in which case we just assume it to be true. If $\mathcal{X}$ is the set of parameters, or meta-variables, that occur in this proof rule, then the last proof step in the derivation of $C \Downarrow R$ consists of an instance of the parameters in $\mathcal{X}$ that yields corresponding instances $(C, R, C'_1, R'_1, \ldots, C'_m, R'_m, \text{true})$ of, respectively, $(C_0, R_0, C_1, R_1, \ldots, C_m, R_m, \text{condition})$, such that $\text{BigStep} \vdash C_i' \Downarrow R_i'$ for all $1 \leq i \leq m$; moreover, domain reasoning is allowed at any moment, in particular to evaluate the instance of condition to true. By the (structural) induction hypothesis we have that $R_{\text{BigStep}} \vdash \overline{C}_i' \rightarrow_1 \overline{R}_i'$ for all $1 \leq i \leq m$. More algebraically, the above say that there is a map $\theta : \mathcal{X} \rightarrow T_\Sigma$ such that:

- (domain reasoning) $R_{\text{BigStep}} \vdash \overline{C} = \theta(C_0)$ and $R_{\text{BigStep}} \vdash \overline{R} = \theta(R_0)$;
- (domain reasoning) $R_{\text{BigStep}} \vdash \overline{C}_i' = \theta(C_i')$ and $R_{\text{BigStep}} \vdash \overline{R}_i' = \theta(R_i')$ for all $1 \leq i \leq m$;
- (domain reasoning) $R_{\text{BigStep}} \vdash \theta(\text{condition}) = \text{true}$;
- (induction hypothesis) $R_{\text{BigStep}} \vdash \overline{C}_i' \rightarrow_1 \overline{R}_i'$ for all $1 \leq i \leq m$.

Since $R_{\text{BigStep}}$ contains the rule

\[
(\forall \mathcal{X}) \overline{C}_0 \rightarrow_0 \overline{R}_0 \quad \text{if} \quad \overline{C}_1 \rightarrow_1 \overline{R}_1 \land \overline{C}_2 \rightarrow_2 \overline{R}_2 \land \ldots \land \overline{C}_m \rightarrow_0 \overline{R}_m \land \text{condition},
\]

the Replacement and Equality rules of rewrite logic then give us $R_{\text{BigStep}} \vdash \overline{C} \rightarrow_1 \overline{R}$.

Let us now assume that $R_{\text{BigStep}} \vdash \overline{C} \rightarrow_1 \overline{R}$ and prove that BigStep $\vdash C \Downarrow R$. Since $C$ and $R$ are ground configurations and since all the rewrite rules derived in any given rewrite logic proof contain the same set of variables, it follows that all the rewrite rules derived as part of the proof of $R_{\text{BigStep}} \vdash \overline{C} \rightarrow_1 \overline{R}$ are also ground. We show by structural induction on the rewrite logic derivation tree that any derivable sequent $R_{\text{BigStep}} \vdash u \rightarrow_1 v$ with $u$ and $v$ ground is of the form $R_{\text{BigStep}} \vdash \overline{C}_u \rightarrow_1 \overline{R}_v$, where $C_u$ is a configuration and $R_v$ a result configuration in BigStep such that BigStep $\vdash C_u \Downarrow R_v$. There are two cases to analyze:
• The last step used the Equality proof rule of rewrite logic, that is, $R_{\text{BioStep}} \vdash u = u', R_{\text{BioStep}} \vdash u' \rightarrow v'$, and $R_{\text{BioStep}} \vdash v' = v$. Then by the induction hypothesis there is a configuration $C_{u'}$ and a result configuration $R_{v'}$ such that $u' = \overline{C_{u'}}$, $v' = \overline{R_{v'}}$ and $\text{BioStep} \vdash C_{u'} \downarrow R_{v'}$. Since $R_{\text{BioStep}} \vdash u = u'$ and $R_{\text{BioStep}} \vdash v' = v$, and since equational deduction preserves the sorts of the terms proved equal, it follows that $u$ and $v$ are also of configuration sort. Then let $C_u$ and $R_v$ be the configurations corresponding to $u$ and $v$, respectively, that is, $u = \overline{C_u}$ and $v = \overline{R_v}$. Since domain properties can be tacitly used in big-step derivations, we conclude that $\text{BigStep} \vdash C_u \downarrow R_v$ and $R_v$ is a result configuration.

• The last step is an instance of the Replacement rule of rewrite logic, with the rewrite rule

$$(\forall X) \overline{C_0} \rightarrow \overline{R_0} \text{ if } \overline{C_1} \rightarrow \overline{R_1} \land \overline{C_2} \rightarrow \overline{R_2} \land \ldots \land \overline{C_m} \rightarrow \overline{R_m} \land \text{condition}$$

Then there is some ground substitution $\theta : X \rightarrow T_\Sigma$ and ground terms $u_1, v_1, \ldots, u_m, v_m$ such that $u = \theta(C_0)$ and $v = \theta(R_0)$, $\theta($condition$) = \text{true}$, and $u_i = \theta(C_i)$ and $v_i = \theta(R_i)$ for all $1 \leq i \leq m$. By the induction hypothesis, there are configurations $C_{u_1}, \ldots, C_{u_m}$ and result configurations $R_{v_1}, \ldots, R_{v_m}$ such that $u_i = \overline{C_{u_i}}$, $v_i = \overline{R_{v_i}}$, and $\text{BioStep} \vdash C_{u_i} \downarrow R_{v_i}$ for all $1 \leq i \leq m$. Let $C_u$ and $R_v$ be the configurations corresponding to $u$ and $v$, respectively, that is, $u = \overline{C_u}$ and $v = \overline{R_v}$. We have thus found instances $(C_u, R_u, C_{u_1}, R_{v_1}, \ldots, C_{u_m}, R_{v_m}, \text{true})$ of $(C_0, R_0, C_1, R_1, \ldots, C_m, R_m, \text{condition})$, such that $\text{BioStep} \vdash C_{u_i} \downarrow R_{v_i}$ for all $1 \leq i \leq m$. In other words, we constructed an instance of the $\text{BigStep}$ rule

$$\overline{C_1} \downarrow R_1 \quad \ldots \quad \overline{C_m} \downarrow R_m$$

if condition

with derivations for all its premises. Hence, $\text{BigStep} \vdash C \downarrow R$.

\[\square\]

The non-nestedness assumption on configurations in $\text{BigStep}$ guarantees that the resulting rewrite rules in $R_{\text{BioStep}}$ only apply at the top of the term they rewrite. The irreducibility of the result configurations guarantees that $R_{\text{BioStep}}$ does not do more rewrite steps than intended, because rewriting is an inherently transitively closed relation, while the big-step relation $\downarrow$ is not.

Since one typically perceives parameters as variables anyway, the only apparent difference between $\text{BigStep}$ and $R_{\text{BioStep}}$ is the different notational conventions they use ($\rightarrow$ instead of $\downarrow$ and conditional rewrite rules instead of conditional deduction rules). As Theorem 13 shows, there is a one-to-one correspondence also between their corresponding “computations” (or executions, or derivations). Therefore, $R_{\text{BioStep}}$ is the big-step operational semantics $\text{BioStep}$, and not an encoding of it.

At our knowledge, there is no rewrite engine\footnote{Maude’s rewrite\cite{DBLP:conf/models/1995} command does not inhibit the transitive closure of the rewrite relation, it only stops the rewrite engine on a given term after one application of a rule on that term; however, many (transitive) applications of rules are allowed when solving the condition of that rule.} that supports the one-step rewrite relation $\rightarrow^1$ (that appears in Theorem 13). Indeed, rewrite engines aim at high-performance implementations of the general rewrite relation $\rightarrow$, which may even involve parallel rewriting (see Section 2.5 for the precise definitions of $\rightarrow^1$ and $\rightarrow$); $\rightarrow^1$ is meaningful only from a theoretical perspective and there is little to no practical motivation for an efficient implementation of it. Therefore, in order to execute the rewrite theory $R_{\text{BioStep}}$ resulting from the mechanical translation of big-step semantics $\text{BioStep}$, one needs to take some precautions to ensure that $\rightarrow^1$ is actually identical to $\rightarrow$.

Given any rewrite theory $\mathcal{R}$, a sufficient condition for $\rightarrow^1$ to be the same as $\rightarrow$ in $\mathcal{R}$ is for the right-hand-sides of the rules in $\mathcal{R}$ to generate terms which make any context that contains them unmatchable by any rule
in $\mathcal{R}$. Fortunately, the two assumptions we made about our original BigStep semantics guarantee this property. First, the non-nestedness assumption guarantees that configurations can only appear at the top of a term, so the only contexts that contain configurations are the configurations themselves. Second, the irreducibility of result configurations assumption ensures that the rules in $\mathcal{R}_{\text{BigStep}}$ will never match any result configurations. Therefore, we can conclude the following important

**Corollary 3.** Under the same hypotheses and assumptions as in Theorem 13

$$\text{BigStep} \vdash C \downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow^1 \overline{R} \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow \overline{R}.$$

Our current BigStep(IMP) semantics verifies our second assumption, since the configurations to the left of $\downarrow$ and the result configurations to the right of $\downarrow$ are always distinct. Unfortunately, in general that may not always be the case. For example, when we extend IMP with side effects in Section 5.10, the state also needs to be part of result configurations, so the semantics of integers is going to be given by an unconditional rule of the form $(i, \sigma) \downarrow (i, \sigma)$, which after translation becomes the rewrite rule $(i, \sigma) \rightarrow (i, \sigma)$. This rule will make the rewrite relation $\rightarrow$ not terminate anymore (although the relation $\rightarrow^1$ will still terminate). There are at least two simple ways to ensure the irreducibility of result configurations, and thus make Corollary 3 still hold:

1. It is highly expected that the only big-step rules in BigStep having a result configuration to the left of $\downarrow$ are unconditional rules of the form $R \downarrow R$; such rules typically say that a value is already evaluated. If that is the case, then one can simply drop all the corresponding rules $R \rightarrow R$ from $\mathcal{R}_{\text{BigStep}}$ and the resulting rewrite theory, say $\mathcal{R}'_{\text{BigStep}}$ still has the property BigStep $\vdash C \downarrow R \iff \mathcal{R}'_{\text{BigStep}} \vdash \overline{C} \rightarrow \overline{R}$, which is desirable in order to execute the big-step definition on rewrite engines, although the property BigStep $\vdash C \downarrow R \iff \mathcal{R}'_{\text{BigStep}} \vdash \overline{C} \rightarrow^1 \overline{R}$ will not hold anymore, because, e.g., even though $R \downarrow R$ is a rule in BigStep, it is not the case that $\mathcal{R}'_{\text{BigStep}} \vdash \overline{R} \rightarrow^1 \overline{R}$.

2. If BigStep contains pairs $R' \downarrow R$ where $R'$ and $R$ are possibly different result configurations, then one can apply the following general procedure. Change or augment the syntax of the configurations to the left or to the right of $\downarrow$, so that those changed or augmented configurations will always be different from the other ones. This is the technique employed in our representation of small-step operational semantics in rewriting logic in Section 5.3. More precisely, we prepend all the configurations to the left of the rewrite relation in $\mathcal{R}_{\text{BigStep}}$ with a circle $\circ$, e.g., $\circ C \rightarrow R$, with the intuition that the circled configurations are active, while the other ones are inactive.

Regardless of how the desired property BigStep $\vdash C \downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow \overline{R}$ is ensured, note that, unfortunately, $\mathcal{R}_{\text{BigStep}}$ lacks the main strengths of rewrite logic that make it a good formalism for concurrency: in rewrite logic, rewrite rules can apply under any context and in parallel. Indeed, the rules of $\mathcal{R}_{\text{BigStep}}$ can only apply at the top, sequentially.

**Big-Step SOS of IMP in Rewrite Logic**

Figure 3.8 shows the rewrite logic theory $\mathcal{R}_{\text{BigStep}(IMP)}$ that is obtained by applying the procedure above to the big-step semantics of IMP, BigStep(IMP), in Figure 3.7. We have used the rewrite logic convention that variables start with upper-case letters. For the state variable, we used $\sigma$, that is, a larger $\sigma$ symbol.

Note how the three side conditions that appear in the proof system in Figure 3.7 turned into normal conditions of rewrite rules. In particular, the two side conditions saying that $\sigma(x)$ is defined became the algebraic term $\sigma(X) \neq \bot$ of Boolean sort.

The following corollary of Theorem 13 and Corollary 3 establishes the faithfulness of the representation of the big-step operational semantics of IMP in rewrite logic:

127
The obtained IMP interpreter actually has acceptable performance; for example, all the programs in Figure 3.4 together take a fraction of a second to execute on conventional PCs or laptops.

In fact, Maude needs only one rewrite logic step to rewrite any configuration; in particular,

```maude
```

Figure 3.8: $\mathcal{R}_{\text{BigStep}(\text{IMP})}$: the big-step SOS of IMP in rewrite logic.

**Corollary 4.** $\text{BigStep}(\text{IMP}) \vdash C \downarrow R \iff \mathcal{R}_{\text{BigStep}(\text{IMP})} \vdash \overline{C} \rightarrow \overline{R}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system $\text{BigStep}(\text{IMP})$ and generic rewrite logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{BigStep}(\text{IMP})}$, so the two are faithfully equivalent.

★ **Maude Definition of IMP Big-Step SOS**

Figure 3.9 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{BigStep}(\text{IMP})}$ in Figure 3.8 including a representation of the algebraic signature in Figure 3.6 for configurations as needed for big-step SOS. The Maude module IMP-SEMANTICS-BIGSTEP in Figure 3.9 is executable, so Maude, through its rewriting capabilities, yields an interpreter for IMP; for example, the command

```maude
rewrite < sumPgm > .
```

where `sumPgm` is the first program defined in the module IMP-PROGRAMS in Figure 3.4 produces a result of the form (the exact statistics are irrelevant here, so they were replaced by “...”):

```
rewrites: 5218 in ... cpu (... real) (... rewrites/second)
result Configuration: < n |-> 0 , s |-> 5050 >
```

The obtained IMP interpreter actually has acceptable performance; for example, all the programs in Figure 3.4 together take a fraction of a second to execute on conventional PCs or laptops.
mod IMP-CONFIGURATIONS-BIGSTEP is including IMP-SYNTAX + STATE.

sort Configuration.
op <_,_> : AExp State -> Configuration.
op <_> : Int -> Configuration.
op <_,_> : BExp State -> Configuration.
op <_> : Bool -> Configuration.
op <_,_> : Stmt State -> Configuration.
op <_> : State -> Configuration.
op <_> : Pgm -> Configuration.
endm

mod IMP-SEMANTICS-BIGSTEP is including IMP-CONFIGURATIONS-BIGSTEP.

var X : Id . var Xl : List{Id} . var Sigma Sigma' Sigma1 Sigma2 : State.
var I I1 I2 : Int . var T : Bool .
var A A1 A2 : AExp . var B B1 B2 : BExp . var S S1 S2 : Stmt .

rl < I,Sigma > => < I > .
crl < X,Sigma > => < Sigma(X) >
if Sigma(X) =/=Bool undefined .
crl < A1 + A2,Sigma > => < I1 + Int I2 >
if < A1,Sigma > => < I1 > /
< A2,Sigma > => < I2 > .
crl < A1 / A2,Sigma > => < I1 / Int I2 >
if < A1,Sigma > => < I1 > /
< A2,Sigma > => < I2 > /
I2 =/=Bool 0 .
rl < T,Sigma > => < T > .
crl < A1 <= A2,Sigma > => < I1 <= Int I2 >
if < A1,Sigma > => < I1 > /
< A2,Sigma > => < I2 > .
crl < ! B,Sigma > => < false >
if < B,Sigma > => < true > .
crl < ! B,Sigma > => < true >
if < B,Sigma > => < false > .
crl < B1 && B2,Sigma > => < false >
if < B1,Sigma > => < false > .
crl < B1 && B2,Sigma > => < T >
if < B1,Sigma > => < true > /
< B2,Sigma > => < T > .

rl < {},Sigma > => < Sigma > .
crl < {S},Sigma > => < Sigma' >
if < S,Sigma > => < Sigma' > .
crl < X = A ;,Sigma > => < Sigma[I / X] >
if < A,Sigma > => < I > /
Sigma(X) =/=Bool undefined .
crl < S1 S2,Sigma > => < Sigma2 >
if < S1,Sigma > => < Sigma1 > /
< S2,Sigma1 > => < Sigma2 > .
crl < if (B) S1 else S2,Sigma > => < Sigma1 >
if < B,Sigma > => < true > /
< S1,Sigma > => < Sigma1 > .
crl < if (B) S1 else S2,Sigma > => < Sigma2 >
if < B,Sigma > => < false > /
< S2,Sigma > => < Sigma2 > .
crl < while (B) S,Sigma > => < Sigma >
if < B,Sigma > => < false > .
crl < while (B) S,Sigma > => < Sigma' >
if < B,Sigma > => < true > /
< while (B) S,Sigma > => < Sigma' > .
crl < int Xl ; S > => < Sigma >
if < S,(Xl |-> 0) > => < Sigma > .
endm

Figure 3.9: The big-step SOS of IMP in Maude, including the definition of configurations.
will give the same output as above. Recall from Section 2.5.6 that Maude performs a potentially exhaustive search to satisfy the rewrites in rule conditions. Thus, a large number of rule instances can be attempted in order to apply one conditional rule, so a \texttt{rewrite} [1] command can take a long time; it may not even terminate. Nevertheless, thanks to Theorem 13, Maude’s implicit search mechanism in conditions effectively achieves a proof searcher for big-step SOS derivations.

Once one has a rewrite logic definition in Maude, one can use any of the general-purpose tools provided by Maude on that definition; the rewrite engine is only one of them. For example, one can exhaustively search for all possible behaviors of a program using the \texttt{search} command:

\begin{verbatim}
search < sumPgm > =>! Cfg:Configuration .
\end{verbatim}

Since our IMP language so far is deterministic, the \texttt{search} command will not discover any new behaviors. In fact, the \texttt{search} command will only discover two configurations in total, the original configuration < \texttt{sumPgm} > and the result configuration < \texttt{n |-> 0 & s |-> 5050} >. However, as shown in Section 3.5 where we extend IMP with various language features, the \texttt{search} command can indeed show all the behaviors of a non-deterministic program (restricted only by the limitations of the particular semantic style employed).

### 3.2.4 Defining a Type System for IMP Using Big-Step SOS

Big-step SOS is routinely used to define type systems for programming languages, even though in most cases this connection is not made explicit. In this section we demonstrate the use of big-step SOS for defining a type system for IMP, following the same steps as above but more succinctly. Type systems is a broad subject, with many variations and important applications to programming languages. Our intention in this section is twofold: on the one hand we show that big-step SOS is not limited to only defining language semantics, and, on the other hand, we introduce the reader to type systems by means of a very simple example.

The idea underlying big-step SOS definitions of type systems is that a given program or fragment of program in a given type environment reduces, in one big step, to its type. Like states, type environments are also partial mappings, but from variable names into types instead of values. A common notation for a type judgment is $\Gamma \vdash c : \tau$, where $\Gamma$ is a type environment, $c$ is a program or fragment, and $\tau$ is a type. This type judgment reads “in type environment $\Gamma$, program or fragment $c$ has type $\tau$”. One can find countless variations of the notation for type judgments in the literature, usually adding more items (pieces of information) to the left of $\vdash$, to its right, or as subscripts or superscripts of it. There is, unfortunately, no well-established notation for all type judgments. Nevertheless, type judgments are special big-step sequents relating two special configurations, one including the givens and the other the results. For example, a simple type judgment $\Gamma \vdash c : \tau$ like above can be regarded as a big-step sequent ($c, \Gamma$) $\Downarrow$ ($\tau$). However, this notation is not preferred.

Figure 3.10 depicts our type system for IMP, which is a nothing but a big-step SOS proof system. We, however, follow the more conventional notation for type judgments discussed above, with one slight change: since in IMP variables are intended to hold only integer values, there is no need for type environments; instead, we replace them by lists of variables, each meant to have the type \texttt{int}. Therefore, $x:l \vdash c : \tau$ with $c$ and $\tau$ as above but with $x:l$ a list of variables reads as follows: “when the variables in $x:l$ are defined, $c$ has the type $\tau$”. We drop the list of variables from the typing judgments of programs, because that would be empty anyway. The big-step SOS rules in Figure 3.10 define the typing policy of each language construct of IMP, guaranteeing all together that a program $p$ types, that is, that $\vdash p : \texttt{pgm}$ is derivable if and only if each construct is used according to its intended typing policy and, moreover, that $p$ declares each variable that it uses. For our simple IMP language, a CFG parser using the syntax defined in Figure 3.1 would already guarantee that each construct is used as intended. Note, however, that the second desired property of our type system (each used variable is declared) is context dependent.
Figure 3.10: BigStepTypeSystem_IMP — Type system of IMP using big-step SOS ($xl \in \text{List}(Id); \ i \in \text{Int}; x \in \text{Id}; a, a_1, a_2 \in \text{AExp}; b, b_1, b_2 \in \text{BExp}; s, s_1, s_2 \in \text{Stmt}$).
Let us next use the type system in Figure 3.10 to type the program sumPgm in Figure 3.4. We split the proof tree in proof subtrees. Note first that using the rules (BigStepTypeSystem-Int) (first level), (BigStepTypeSystem-Asgn) (second level) and (BigStepTypeSystem-Seq) (third level), we can derive the following proof tree, say $\text{tree}_1$:

$$\text{tree}_1 = \left\{ \begin{array}{l}
n,s \vdash 100 : \text{int} \\
n,s \vdash 0 : \text{int} \\
\frac{n,s \vdash (n = 100)}{n,s \vdash (s = 0)} : \text{stmt} \\
\frac{n,s \vdash (s = 0)}{n,s \vdash (n = 100; s = 0)} : \text{stmt} \end{array} \right\}$$

Similarly, using rules (BigStepTypeSystem-Lookup) and (BigStepTypeSystem-Int) (first level), (BigStepTypeSystem-Leq) (second level), and (BigStepTypeSystem-Not) (third level), we can derive the following proof tree, say $\text{tree}_2$:

$$\text{tree}_2 = \left\{ \begin{array}{l}
n,s \vdash n : \text{int} \\
n,s \vdash 0 : \text{int} \\
\frac{n,s \vdash (n \leq 0)}{n,s \vdash (! (n \leq 0))} : \text{bool} \end{array} \right\}$$

Similarly, we can derive the following proof tree, say $\text{tree}_3$:

$$\text{tree}_3 = \left\{ \begin{array}{l}
n,s \vdash s : \text{int} \\
n,s \vdash s : \text{int} \\
n,s \vdash n : \text{int} \\
n,s \vdash -1 : \text{int} \\
\frac{n,s \vdash (s + n)}{n,s \vdash (s = s + n)} : \text{stmt} \\
\frac{n,s \vdash (n - 1)}{n,s \vdash (n = n - 1)} : \text{stmt} \\
\frac{n,s \vdash (s = s + n; n = n + -1)}{n,s \vdash \{ s = s + n; n = n + -1; \}} : \text{block} \end{array} \right\}$$

Finally, we can now derive the tree that proves that sumPgm is well-typed:

$$\text{tree}_1 \text{tree}_2 \text{tree}_3 \\
n,s \vdash (\text{while} \ (! (n \leq 0)) \{ s = s + n; n = n + -1; \}) : \text{stmt} \\
n,s \vdash (\text{if} n = 100; \ s = 0; \ \text{while} \ (! (n \leq 0)) \{ s = s + n; n = n + -1; \}) : \text{stmt} \\
\vdash (\text{int} n; n = 100; \ s = 0; \ \text{while} \ (! (n \leq 0)) \{ s = s + n; n = n + -1; \}) : \text{pgm}$$

A major role of a type system is to filter out a set of programs which are obviously wrong. Unfortunately, it is impossible to filter out precisely those programs which would execute erroneously. For example, note that a division is considered type safe whenever its two arguments type to integers, but no check is being made on whether the denominator is 0 or not. Indeed, statically checking whether an expression has a certain value at a certain point in a program is an undecidable problem. Also, no check is being made on whether a detected type error is reachable or not (if unreachable, the detected type error will never show up at runtime). Statically checking whether a certain point in a program is reachable is also an undecidable problem. One should therefore be aware of the fact that in general a type system may allow programs which run into errors when executed and, moreover, that it may reject programs which would execute correctly.
\[ (I, XI) \rightarrow \text{(int)} \]
\[ (X, (XI, X', XI')) \rightarrow \text{(int)} \]
\[ (A_1 + A_2, XI) \rightarrow \text{(int)} \text{ if } (A_1, XI) \rightarrow \text{(int)} \land (A_2, XI) \rightarrow \text{(int)} \]
\[ (\sim \text{IMPA}, A_2, XI) \rightarrow \text{(int)} \text{ if } (A_1, XI) \rightarrow \text{(int)} \land (A_2, XI) \rightarrow \text{(int)} \]
\[ (A_1 <= A_2, XI) \rightarrow \text{(bool)} \text{ if } (A_1, XI) \rightarrow \text{(int)} \land (A_2, XI) \rightarrow \text{(int)} \]
\[ (T, XI) \rightarrow \text{(bool)} \]
\[ (\{ B, XI \} \rightarrow \text{(bool)} \text{ if } (B, XI) \rightarrow \text{bool} \]
\[ (B_1 \&\& B_2, XI) \rightarrow \text{(bool)} \text{ if } (B_1, XI) \rightarrow \text{bool} \land (B_2, XI) \rightarrow \text{bool} \]
\[ (\{ \}, XI) \rightarrow \text{(block)} \]
\[ (\{ S \}, XI) \rightarrow \text{(block)} \text{ if } (S, XI) \rightarrow \text{(ST)} \]
\[ \langle X = A, (XI, X', XI') \rangle \rightarrow \text{(stmt)} \text{ if } \langle A, (XI, X', XI') \rangle \rightarrow \text{(int)} \]
\[ \langle S_1 S_2, XI \rangle \rightarrow \text{(stmt)} \text{ if } \langle S_1, XI \rangle \rightarrow \text{(ST\{1\})} \land \langle S_2, XI \rangle \rightarrow \text{(ST\{2\})} \]
\[ \langle \text{if} (B) S_1 \text{else} S_2, XI \rangle \rightarrow \text{(stmt)} \text{ if } \langle B, XI \rangle \rightarrow \text{bool} \land \langle S_1, XI \rangle \rightarrow \text{(block)} \land \langle S_2, XI \rangle \rightarrow \text{(block)} \]
\[ \langle \text{while} (B) S, XI \rangle \rightarrow \text{(stmt)} \text{ if } \langle B, XI \rangle \rightarrow \text{bool} \land \langle S, XI \rangle \rightarrow \text{(block)} \]
\[ \langle \text{int} XI; S \rangle \rightarrow \text{(pgm)} \text{ if } \langle S, XI \rangle \rightarrow \text{(ST)} \]

Figure 3.11: \( R_{\text{BigStepTypeSystem_IMP}} \): type system of IMP using big-step SOS in rewrite logic. Assume a sort Type with constants int, bool, and pgm, and a subsort of it StmtType with constants block and stmt. The sorts of the involved variables is understood (from Figure 3.10), except for \( T \) which has the sort Bool and for ST, ST\{1\} and ST\{2\} which have the sort StmtType.

Figure 3.11 shows a translation of the big-step SOS in Figure 3.10 into a rewrite logic theory, following the general technique described in Section 3.2.3 but with two simple optimizations explained shortly. This translation is based on the reinterpretation of type judgments as big-step SOS sequents mentioned above. The following configurations are used in the rewrite theory in Figure 3.11:

- \( \langle A, XI \rangle \) grouping arithmetic expressions \( A \) and variable lists \( XI \);
- \( \langle B, XI \rangle \) grouping Boolean expressions \( B \) and variable lists \( XI \);
- \( \langle S, XI \rangle \) grouping statements \( S \) and variable lists \( XI \);
- \( \langle P \rangle \) holding programs \( P \);
- \( \langle \tau \rangle \) holding types \( \tau \), which can be int, bool, block, stmt, or pgm.

The two optimizations refer to how to test the various memberships that occur in some of the rules’ side conditions. One optimization is to use associative matching to check membership to a list; this applies to the rules (BigStepTypeSystem-Lookup) and (BigStepTypeSystem-Assn). Another optimization is to use a (sub)sort as an umbrella for a set of terms of interest (constants in our case); for example, we let \( T \) be a variable of sort Bool (i.e., \( T \) can match both true and false) when defining the rewrite rule corresponding to (BigStepTypeSystem-Bool), and we let ST, ST\{1\}, and ST\{2\} be variables ranging over a new subsort StmtType of Type when defining the rewrite rules corresponding to (BigStepTypeSystem-Block), (BigStepTypeSystem-Seq) and (BigStepTypeSystem-Pgm).

By Corollary 3.3, we have that a program \( p \) is well-typed, that is, \( \vdash p : \text{pgm} \) is derivable with the proof system in Figure 3.10 if and only if \( R_{\text{BigStepTypeSystem_IMP}} \vdash \langle p \rangle \rightarrow \text{pgm} \).
mod IMP-TYPES is
  sorts StmtType Type .
  subsort StmtType < Type .
  ops block stmt : -> StmtType .
  ops int bool pgm : -> Type .
endm

mod IMP-TYPE-SYSTEM-CONFIGURATIONS-BIGSTEP is including IMP-SYNTAX + IMP-TYPES .
  sort Configuration .
  op <_,_> : AExp List{Id} -> Configuration .
  op <_,_> : BExp List{Id} -> Configuration .
  op <_,_> : Stmt List{Id} -> Configuration .
  op <_> : Pgm -> Configuration .
  op <_> : Type -> Configuration .
endm

mod IMP-TYPE-SYSTEM-BIGSTEP is including IMP-TYPE-SYSTEM-CONFIGURATIONS-BIGSTEP .
  var X : Id .
  var Xl Xl' : List{Id} .
  var I : Int .
  var T : Bool .
  var A A1 A2 : AExp .
  var B B1 B2 : BExp .
  var S S1 S2 : Stmt .
  var ST ST1 ST2 : StmtType .

  rl < I,Xl > => < int > .
  rl < X,(Xl,X,Xl') > => < int > .
  crl < A1 + A2,Xl > => < int >
    if < A1,Xl > => < int > /
     < A2,Xl > => < int > .
  crl < A1 / A2,Xl > => < int >
    if < A1,Xl > => < int > /
     < A2,Xl > => < int > .

  rl < T,Xl > => < bool > .
  crl < A1 <= A2,Xl > => < bool >
    if < A1,Xl > => < int > /
     < A2,Xl > => < int > .
  crl < ! B,Xl > => < bool >
    if < B,Xl > => < bool > .
  crl < B1 && B2,Xl > => < bool >
    if < B1,Xl > => < bool > /
     < B2,Xl > => < bool > .

  rl < {},Xl > => < block > .
  crl < {S},Xl > => < block >
    if < S,Xl > => < ST > .
  crl < X = A ;,(Xl,X,Xl') > => < stmt >
    if < A,(Xl,X,Xl') > => < int > .
  crl < S1 S2,Xl > => < stmt >
    if < S1,Xl > => < ST1 > /
     < S2,Xl > => < ST2 > .
  crl < if (B) S1 else S2,Xl > => < stmt >
    if < B,Xl > => < bool > /
     < S1,Xl > => < block > /
     < S2,Xl > => < block > .
  crl < while (B) S,Xl > => < stmt >
    if < B,Xl > => < bool > /
     < S,Xl > => < block > .

  crl < int Xl ; S > => < pgm >
    if < S,Xl > => < ST > .
endm

Figure 3.12: The type-system of IMP using big-step SOS in Maude, including the definition of types and configurations.
Maude Definition of a Type System for IMP using Big-Step SOS

Figure 3.12 shows the Maude representation of the rewrite theory $R_{\text{BigStepTypeSystem(IMP)}}$ in Figure 3.11 including a representation of the algebraic signature for the needed configurations. The Maude module IMP-TYPE-SYSTEM-BIGSTEP in Figure 3.12 is executable, so Maude, through its rewriting capabilities, yields a type checker for IMP; for example, the command

```
rewrite < sumPgm > .
```

where `sumPgm` is the first program defined in the module IMP-PROGRAMS in Figure 3.4 produces a result of the form (the exact statistics are irrelevant here, so they were replaced by “…”):

```
rewrites: 20 in ... cpu (... real) (... rewrites/second)
result Configuration: < pgm >
```

A type system is generally expected to be deterministic. Nevertheless, implementations of it (particularly rewrite-based ones) may mistakenly be non-deterministic (non-confluent; see Section 2.1.4). To gain confidence in the determinism of the Maude definition in Figure 3.12 one may exhaustively search for all possible behaviors yielded by the type checker:

```
search < sumPgm > =>! Cfg:Configuration .
```

As expected, this finds only one solution. This Maude definition of IMP’s type checker is very simple and one can easily see that it is confluent (it is orthogonal—see Section 2.1.4), so the search is redundant. However, the search command may be useful for testing more complex type systems.

3.2.5 Notes

Big-step structural operational semantics (big-step SOS) was introduced under the name natural semantics by Kahn [34] in 1987. Even though he introduced it in the limited context of defining Mini-ML, a simple pure (no side effects) version of the ML language, Kahn’s aim was to propose natural semantics as a “unified manner to present different aspects of the semantics of programming languages, such as dynamic semantics, static semantics and translation” (Section 1.1 in [34]). Kahn’s original notation for big-step sequents was $\sigma \vdash a \Rightarrow i$, with the meaning that $a$ evaluates to $i$ in state (or environment) $\sigma$. Kahn, like many others after him (including ourselves; e.g., Section 3.2.4), took the freedom to using a different notation for type judgments, namely $\Gamma \vdash c : \tau$, where $\Gamma$ is a type environment, $c$ is a program or fragment of program, and $\tau$ is a type. This colon notation for type judgments was already established in 1987; however, Kahn noticed that the way type systems were defined was a special instance of a more general schema, which he called natural semantics (and which is called big-step SOS here and in many other places). Big-step SOS is very natural when defining pure, deterministic and structured languages, so it quickly became very popular. However, Kahn’s terminology for “natural” semantics was inspired from its reminiscence to “natural deduction” in mathematical logic, not necessarily from the fact that it is natural to use. He demonstrated how one can use big-step SOS to define all these in his seminal paper [34], using the Mini-ML language.

As Kahn himself acknowledged, the idea of using proof systems to capture the operational semantics of programming languages goes back to Plotkin [60, 61] in 1981. Plotkin was the first to coin the terminology structural operational semantics (SOS), but what he meant by that was mostly what we call today small-step structural operational semantics (small-step SOS). Note, however, that Plotkin in fact used a combination of small-step and big-step SOS, without calling them as such, using the $\rightarrow$ arrow for small-steps and its transitive closure $\rightarrow^*$ for big-steps. We will discuss small-step SOS in depth in Section 3.3. Kahn and others found big-step SOS more natural and convenient than Plotkin’s SOS, essentially because it is more abstract.
and denotational in nature (which may help in formal reasoning), and one needs fewer rules to define a language semantics.

One of the most notable uses of natural semantics is the formal semantics of Standard ML by Milner et al. \[49\]. Several types of big-step sequents were used in \[49\], such as \( \rho \vdash p \Rightarrow v/f \) for "in environment \( \rho \), sentence \( p \) either evaluates to value \( v \) or otherwise an error or failure \( f \) takes place", and \( \sigma, \rho \vdash p \Rightarrow v, \sigma' \) for "in state \( \sigma \) and environment \( \rho \), sentence \( p \) evaluates to \( v \) and the resulting state is \( \sigma' \)”, and \( \rho, v \vdash m \Rightarrow v'/f \) for "in environment \( \rho \), a match \( m \) either evaluates to \( v' \) or otherwise failure \( f' \)”, among many others. After more than twenty years of natural semantics, it is now common wisdom that big-step semantics is inappropriate as a rigorous formalism for defining languages with complex features such as exceptions or concurrency. To give a reasonably compact and readable definition of Standard ML in \[49\], Milner et al. had to make several informal notational conventions, such as a "state convention" to avoid having to mention the state in every rule, and an "exception convention" to avoid having to more than double the number of rules for the sole purpose of supporting exceptions. As rightfully noticed by Moses \[53\], such conventions are not only ad hoc and language specific, but may also lead to erroneous definitions. Section \[3.5\] illustrates in detail the limitations of big-step operational semantics, both with respect to its incapacity of defining certain rather simple language features and with respect to inconvenience in using it (for example, due to its lack of modularity). One of the common uses of natural semantics these days is to define static semantics of programming languages and calculi, such as type systems (see Section \[3.2.4\]).

Hennessy \[32\] (1990) and Winskel \[87\] (1993) are perhaps the first textbooks proposing big-step SOS in teaching programming language semantics. They define big-step SOS for several simple languages, including ones similar to the IMP language presented in this chapter. Hennessy \[32\] defines languages incrementally, starting with a small core and then adding new features one by one, highlighting a major problem with big-step SOS: its lack of modularity. Indeed, the big-step SOS of a language is entirely redefined several times in \[32\] as new features are added to the language, because adding new features requires changes in the structure of judgments. For example, some big-step SOS judgments in \[32\] evolve from \( a \Rightarrow i \), to \( \sigma \vdash a \Rightarrow i \), to \( D, \sigma \vdash i \) during the language design experiment, where \( a \) is an expression, \( i \) an integer, \( \sigma \) a state (or environment), and \( D \) a set of function definitions.

While the notations of Hennessy \[32\] and of Milner et al. \[49\] are somehow reminiscent of original Kahn’s notation, Winskel \[87\] uses a completely different notation. Specifically, he prefers to use big-step sequents of the form \( \langle a, \sigma \rangle \rightarrow i \) instead of \( \sigma \vdash a \Rightarrow i \). There seems to be, unfortunately, no uniform and/or broadly accepted notation for SOS sequents in the literature, be they big-step or small-step. As already explained earlier in this section, for the sake of uniformity at least throughout this book, we will make an effort to consider sequents of the form \( C \parallel R \) in our big-step SOS definitions, where \( C \) and \( R \) are configurations. Similarly, we will make an effort to use the notation \( C \rightarrow C' \) for small-step sequents (see Section \[3.3\]). We will make it explicit when we deviate from our uniform notation, explaining how the temporary notation relates to the uniform one, as we did in Section \[3.2.4\].

Big-step SOS is the semantic approach which is probably the easiest to implement in any language or to represent in any computational logic. There are countless approaches to implementing or encoding big-step SOS in various languages or logics, which we cannot enumerate here. We only mention rewriting-based ones which are directly related to the approach followed in this book. Vardejo and Martí-Oliet \[84\] proposed big-step SOS implementations in Maude for several languages, including Hennessy’s languages \[32\] and Kahn’s Mini-ML \[54\]. Vardejo and Martí-Oliet were mainly interested in demonstrating the strengths of Maude 2 to give executable semantics to concrete languages, rather than in proposing general representations of big-step SOS into rewrite logic that work for any language. Besides Vardejo and Martí-Oliet, several other authors used rewrite logic and Maude to define and implement language semantics for languages or calculi.
following a small-step approach. These are discussed in Section 3.3.4; we here only emphasize that most of those can likely be adapted into big-step SOS definitional or implementation styles, because big-step SOS can be regarded as a special case of small-step SOS, one in which the small step is “big”.

There is a common misconception that big-step SOS is “efficient” when executed. This misconception is fueled by case studies where efficient interpreters for various (deterministic or deterministic fragments of) languages, were more or less mechanically derived from the big-step SOS of those languages. Recall that a big-step SOS is a formal proof system, and that proof systems are typically meant to tell what is possible in the defined language, calculus or system, and not how to implement it. A big-step SOS can indeed be used as a basis to develop efficient interpreters, but one should be aware of the fact that when that happens it happens either because the big-step SOS has a particularly convenient form, where at most one proof rule can apply at any given moment, or because one cuts corners in the implementation by deliberately ignoring the proof search process needed to detect which proof rule applies, and instead arbitrarily picking one matching rule and stopping the execution if its premises cannot be proved. Section 3.5 illustrates a big-step SOS whose rules are not syntax-driven (e.g., expressions have side effects and arithmetic operators are non-deterministic); as discussed there, its faithful Maude implementation is indeed very slow, requiring exponential complexity in some cases to derive the rule premises.

### 3.2.6 Exercises

Prove the following exercises, all referring to the IMP big-step SOS in Figure 3.7.

**Exercise 54.** Change the rule (BigStep-Div) so that division short-circuits when a₁ evaluates to 0. *(Hint: may need to replace it with two rules, like for the semantics of conjunction).*

**Exercise 55.** Change the big-step semantics of the IMP conjunction so that it is not short-circuited.

**Exercise 56.** Add an “error” state and modify the big-step semantics in Figure 3.7 to allow derivations of sequents of the form ⟨s, σ⟩ ⇓ ⟨error⟩ or ⟨p⟩ ⇓ ⟨error⟩ when s evaluated in state σ or when p evaluated in the initial state performs a division by zero.

**Exercise 57.** Prove that the transition relation defined by the BigStep(IMP) proof system in Figure 3.7 is deterministic, that is:

- If BigStep(IMP) ⊢ ⟨a, σ⟩ ⇓ ⟨i⟩ and BigStep(IMP) ⊢ ⟨a, σ⟩ ⇓ ⟨i’⟩ then i = i’;
- If BigStep(IMP) ⊢ ⟨b, σ⟩ ⇓ ⟨t⟩ and BigStep(IMP) ⊢ ⟨b, σ⟩ ⇓ ⟨t’⟩ then t = t’;
- If s terminates in σ then there is a unique σ’ such that BigStep(IMP) ⊢ ⟨s, σ⟩ ⇓ ⟨σ’⟩;
- If p terminates then there is a unique σ such that BigStep(IMP) ⊢ ⟨p⟩ ⇓ ⟨σ⟩.

Prove the same results above for the proof system detecting division-by-zero as in Exercise 56.

**Exercise 58.** Show that there is no algorithm, based on the big-step proof system in Figure 3.7 or on anything else, which takes as input an IMP program and says whether it terminates or not.

---

4We call such big-step SOS definitions syntax-driven.

5Either because one proved that this is sound, or because one’s intention is to trade soundness for performance.
Is this enough? Should I add more citations here? Ask Narciso, Peter and Jose for their and others’ works embedding big-step into rewrite logic.

change \texttt{=/=} \texttt{Bool} in \texttt{=/=} \texttt{Int}, etc.

make sure only the IMP macros are used above, e.g., \texttt{plusIMP}, etc.
3.3 Small-Step Structural Operational Semantics (Small-Step SOS)

Known also under the names transition semantics, reduction semantics, one-step operational semantics, and computational semantics, small-step structural operational semantics, or small-step SOS for short, formally captures the intuitive notion of one atomic computational step. Unlike in big-step SOS where one defines all computation steps in one transition, in a small-step SOS a transition encodes only one computation step. To distinguish small-step from big-step transitions, we use a plain arrow \( \rightarrow \) instead of \( \parallel \). To execute a small-step SOS, or to relate it to a big-step SOS, we need to transitively close the small-step transition relation. Indeed, the conceptual relationship between big-step SOS and small-step SOS is that for any configuration \( C \) and any result configuration \( R \), \( C \parallel R \) if and only if \( C \rightarrow^* R \). Small-step SOS is typically preferred over big-step SOS when defining languages with a high-degree of non-determinism, such as, for example, concurrent languages, because in a small-step semantics one has direct control over what can execute and when.

Like big-step SOS, a small-step SOS of a programming language or calculus is also given as a formal proof system (see Section 2.1.5). The small-step SOS sequents are also binary relations over configurations like in big-step SOS, but in small-step SOS they are written \( C \rightarrow C' \) and have the meaning that \( C' \) is a configuration obtained from \( C \) after one step of computation. A small-step SOS rule therefore has the form

\[
\frac{C_1 \rightarrow C'_1 \quad C_2 \rightarrow C'_2 \quad \ldots \quad C_n \rightarrow C'_n}{C_0 \rightarrow C'} \quad \text{[if condition]}
\]

where \( C_0, C_1, C'_1, C_2, C'_2, \ldots, C_n, C'_n \) are configurations holding fragments of program together with all the needed semantic components, like in big-step SOS, and condition is an optional side condition. Unlike in big-step SOS, the result configurations do not need to be explicitly defined. In small-step SOS they are implicit: they are precisely those configurations which cannot be reduced anymore using the one-step relation.

Given a configuration holding a fragment of program, a small-step of computation typically takes place in some subpart of the fragment. However, when each of the subparts is already reduced, then the small-step can apply on the part itself. A small-step SOS is therefore finer-grain than big-step SOS, and thus more verbose, because one has to cover all the cases where a computation step can take place. For example, the small-step SOS of addition in IMP is

\[
\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a_2, \sigma \rangle}
\]

\[
\frac{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a'_2, \sigma \rangle}
\]

\[
\frac{\langle i_1 + i_2, \sigma \rangle}{\langle i_1 + \text{int} i_2, \sigma \rangle}
\]

Here, the meaning of a relation \( \langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle \) is that arithmetic expression \( a \) in state \( \sigma \) is reduced, in one small-step, to arithmetic expression \( a' \) and the state stays unchanged. Like for big-step SOS, one can encounter various other notations for small-step SOS configurations in the literature, e.g., \( [a, \sigma] \), or \( (a, \sigma) \), or \( \{a, \sigma\} \), or \( \langle a \mid \sigma \rangle \), etc. Like for big-step SOS, we prefer to uniformly use the angle-bracket-and-comma notation \( \langle a, \sigma \rangle \). Also, like for big-step SOS, one can encounter various decorations on the transition arrow \( \rightarrow \), a notable situation being when the transition is labeled. Again like for big-step SOS, we assume that such transition decorations are incorporated in the (source and/or target) configurations. How this can be effectively achieved is discussed in detail in Section 3.6 in the context of modular SOS (which allows rather complex transition labels).

The rules above rely on the fact that expression evaluation in IMP has no side effects. If there were side effects, like in the IMP extension in Section 3.5 then the \( \sigma \)'s in the right-hand-side configurations above
would need to change to a different symbol, say $\sigma'$, to account for the possibility that the small-step in the condition of the rules, and implicitly in their conclusion, may change the state as well. While in big-step SOS it is more common to derive sequents of the form $\langle a, \sigma \rangle \Downarrow \langle i, \sigma \rangle$, in small-step SOS the opposite tends to be norm, that is, it is more common to derive sequents of the form $\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle$ than of the form $\langle a, \sigma \rangle \rightarrow \langle a', \sigma' \rangle$. Nevertheless, the latter sequent type also works when defining languages like IMP whose expressions are side-effect-free (see Exercise 68). Some language designers may prefer this latter style, to keep sequents minimal. However, even if one prefers these simpler sequents, we still keep the angle brackets in the right-hand-sides of the transition relations (for the same reason like in big-step SOS—to maintain a uniform notation); in other words, we write $\langle a, \sigma \rangle \rightarrow \langle a' \rangle$ and not $\langle a, \sigma \rangle \rightarrow a'$.

In addition to rules, a small-step SOS may also include structural identities. For example, we can state that sequential composition is associative using the following structural identity:

$$(s_1 \ s_2) \ s_3 \equiv s_1 \ (s_2 \ s_3)$$

The small-step SOS rules apply modulo structural identities. In other words, the structural identities can be used anywhere in any configuration and at any moment during the derivation process, without counting as computational steps. In practice, they are typically used to rearrange the syntactic term so that some small-step SOS rule can apply. In particular, the structural rule above allows the designer of the small-step SOS to rely on the fact that the first statement in a sequential composition is not a sequential composition, which may simplify the actual SOS rules; this is indeed the case in Section 3.5.4 where we extend IMP with dynamic threads (we do not need structural identities in the small-step SOS definition of the simple IMP language in this section). Structural identities are not easy to execute and/or implement in their full generality, because they can quickly yield an exponential explosion in the number of terms that need to be matched by rules. Their role in SOS is the same as the role of equations in rewriting logic definitions; in fact, we effectively turn them into equations when we embed small-step SOS into rewrite logic (see Section 3.3.3).

### 3.3.1 IMP Configurations for Small-Step SOS

The configurations needed for the small-step SOS of IMP are a subset of those needed for its big-step SOS in Section 3.2.1 Indeed, we still need all the two-component configurations containing a fragment of program and a state, but, for the particular small-step SOS style that we follow in this section, we do not need those result configurations of big-step SOS containing only a value or only a state. If one prefers to instead follow the minimalist style as in Exercise 68, then one would also need the other configuration types. Here are all the configuration types needed for the small-step SOS of IMP given in the remainder of this section:

- $\langle a, \sigma \rangle$ grouping arithmetic expressions $a$ and states $\sigma$;
- $\langle b, \sigma \rangle$ grouping Boolean expressions $b$ and states $\sigma$;
- $\langle s, \sigma \rangle$ grouping statements $s$ and states $\sigma$;
- $\langle p \rangle$ holding programs $p$.

We still need the one-component configuration holding only a program, because we still want to reduce a program in the default initial state (empty) without having to mention the empty state.
Figure 3.13: IMP small-step SOS configurations as an algebraic signature.

**IMP Small-Step SOS Configurations as an Algebraic Signature**

Figure 3.13 shows an algebraic signature defining the IMP configurations above, which is needed for the subsequent small-step operational semantics. We defined this algebraic signature in the same style and following the same assumptions as those for big-step SOS in Section 3.2.1.

### 3.3.2 The Small-Step SOS Rules of IMP

Figures 3.14 and 3.15 show all the rules in our IMP small-step SOS proof system, the former showing the rules corresponding to expressions, both arithmetic and Boolean, and the latter showing those rules corresponding to statements. The sequents that this proof system derives have the forms $\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle$, $\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle$, $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$, and $\langle p \rangle \rightarrow \langle s, \sigma \rangle$, where $a$ ranges over $AExp$, $b$ over $BExp$, $s$ over $Stmt$, $p$ over $Pgm$, and $\sigma$ and $\sigma'$ over $State$.

The meaning of $\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle$ is that given state $\sigma$, the arithmetic expression $a$ reduces in one (small) step to the arithmetic expression $a'$ and the state $\sigma$ stays unchanged. The meaning of $\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle$ is similar but with Boolean expressions instead of arithmetic expressions. The meaning of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ is that statement $s$ in state $\sigma$ reduces in one step to statement $s'$ in a potentially modified state $\sigma'$. The meaning of $\langle p \rangle \rightarrow \langle s, \sigma \rangle$ is that program $p$ reduces in one step to statement $s$ in state $\sigma$ (as expected, whenever such sequents can be derived, the statement $s$ is the body of $p$ and the state $\sigma$ initializes to 0 the variables declared by $p$). The reason for which the state stays unchanged in the sequents corresponding to arithmetic and Boolean expressions is because, as discussed, IMP’s expressions currently have no side effects; we will have to change these rules later on in Section 3.5 when we add a variable increment arithmetic expression construct to IMP. A small-step reduction of a statement may or may not change the state, so we use a different symbol in the right-hand-side of statement transitions, $\sigma'$, to cover both cases.

We next discuss each of the small-step SOS rules of IMP in Figures 3.14 and 3.15. Before we start, note that we have no rules for reducing constant (integer or Boolean) expressions to their corresponding values as we had in big-step SOS (i.e., no rules corresponding to (BigStep-Int) and (BigStep-Bool) in Figure 3.7). Indeed, we do not want to have small-step rules of the form $\langle v, \sigma \rangle \rightarrow \langle v, \sigma \rangle$ because no one-step reductions are further desired on values $v$: adding such rules would lead to undesired divergent SOS reductions later on when we consider the transitive closure of the one-step relation $\rightarrow$. Recall that the big-step SOS relation $\mid\rangle$ captured all the reduction steps at once, including zero steps, and thus it did not need to be transitively closed like the small-step relation $\rightarrow$, so evaluating values to themselves was not problematic in big-step SOS.

The rule (SmallStep-Lookup) happens to be almost the same as in big-step SOS; that’s because variable lookup is an atomic-step operation both in big-step and in small-step SOS. The rules (SmallStep-Add-Arg1), (SmallStep-Add-Arg2), and (SmallStep-Add) give the small-step semantics of addition, and (SmallStep-Div-Arg1), (SmallStep-Div-Arg2), and (SmallStep-Div) that of division, each covering all the three cases where
\[ \langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle \text{ if } \sigma(x) \neq \bot \]  
(SMALL-STEP-LOOKUP)

\[ \langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle \]

\[ \langle a_1 + a_2, \sigma \rangle \rightarrow \langle a_1' + a_2, \sigma \rangle \]  
(SMALL-STEP-ADD-ARG1)

\[ \langle a_2, \sigma \rangle \rightarrow \langle a_2', \sigma \rangle \]

\[ \langle a_1 + a_2, \sigma \rangle \rightarrow \langle a_1 + a_2', \sigma \rangle \]  
(SMALL-STEP-ADD-ARG2)

\[ \langle i_1 + i_2, \sigma \rangle \rightarrow \langle i_1 +_{\text{Int}} i_2, \sigma \rangle \]  
(SMALL-STEP-ADD)

\[ \langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle \]

\[ \langle a_1 / a_2, \sigma \rangle \rightarrow \langle a_1' / a_2, \sigma \rangle \]  
(SMALL-STEP-DIV-ARG1)

\[ \langle a_2, \sigma \rangle \rightarrow \langle a_2', \sigma \rangle \]

\[ \langle a_1 / a_2, \sigma \rangle \rightarrow \langle a_1 / a_2', \sigma \rangle \]  
(SMALL-STEP-DIV-ARG2)

\[ \langle i_1 / i_2, \sigma \rangle \rightarrow \langle i_1 /_{\text{Int}} i_2, \sigma \rangle \text{ if } i_2 \neq 0 \]  
(SMALL-STEP-DIV)

\[ \langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle \]

\[ \langle a_1 <= a_2, \sigma \rangle \rightarrow \langle a_1' <= a_2, \sigma \rangle \]  
(SMALL-STEP-LEQ-ARG1)

\[ \langle a_2, \sigma \rangle \rightarrow \langle a_2', \sigma \rangle \]

\[ \langle i_1 <= a_2, \sigma \rangle \rightarrow \langle i_1 <= a_2', \sigma \rangle \]  
(SMALL-STEP-LEQ-ARG2)

\[ \langle i_1 <= i_2, \sigma \rangle \rightarrow \langle i_1 <=_{\text{Int}} i_2, \sigma \rangle \]  
(SMALL-STEP-LEQ)

\[ \langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle \]

\[ \langle 1 b, \sigma \rangle \rightarrow \langle 1 b', \sigma \rangle \]  
(SMALL-STEP-NOT-ARG)

\[ \langle ! \text{ true}, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle \]  
(SMALL-STEP-NOT-TRUE)

\[ \langle ! \text{ false}, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle \]  
(SMALL-STEP-NOT-FALSE)

\[ \langle b_1, \sigma \rangle \rightarrow \langle b_1', \sigma \rangle \]

\[ \langle b_1 \&\& b_2, \sigma \rangle \rightarrow \langle b_1' \&\& b_2, \sigma \rangle \]  
(SMALL-STEP-AND-ARG1)

\[ \langle \text{false} \&\& b_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle \]  
(SMALL-STEP-AND-FALSE)

\[ \langle \text{true} \&\& b_2, \sigma \rangle \rightarrow \langle b_2, \sigma \rangle \]  
(SMALL-STEP-AND-TRUE)

Figure 3.14: SMALL-STEP(IMP)— Small-step SOS of IMP expressions \((i_1, i_2 \in \text{Int}; x \in \text{Id}; a_1, a_1', a_2, a_2' \in AExp; b, b', b_1, b_1', b_2 \in BExp; \sigma \in \text{State})\).
a small-step reduction can take place. The first two cases in each group may apply non-deterministically. Recall from Section 3.2 that big-step SOS was inappropriate for defining the desired non-deterministic evaluation strategies for + and / . Fortunately, that was not a big problem for IMP, because its intended non-deterministic constructs are side-effect free. Therefore, the intended non-deterministic evaluation strategies of these particular language constructs did not affect the overall determinism of the IMP language, thus making its deterministic (see Exercise 57) big-step SOS definition in Section 3.2 acceptable. As expected, the non-deterministic evaluation strategies of + and / , which this time can be appropriately captured within the small-step SOS, will not affect the overall determinism of the IMP language (that is, the reflexive/transitive closure \( \rightarrow^* \) of \( \rightarrow \); see Theorem 14). These will start making a difference when we add side effects to expressions in Section 3.5.

The rules (SMALLSTEP-LEQ-ARG1), (SMALLSTEP-LEQ-ARG2), and (SMALLSTEP-LEQ) give the deterministic, sequential small-step SOS of \( \leq \). The first rule applies whenever \( a_1 \) is not an integer, then the second rule applies when \( a_1 \) is an integer but \( a_2 \) is not an integer, and finally, when both \( a_1 \) and \( a_2 \) are integers, the third rule applies. The rules (SMALLSTEP-NOT-ARG), (SMALLSTEP-NOT-TRUE), and (SMALLSTEP-NOT-FALSE) are self-explanatory, while the rules (SMALLSTEP-AND-ARG1), (SMALLSTEP-AND-FALSE) and (SMALLSTEP-AND-TRUE) give the short-circuited semantics of and: indeed, \( b_2 \) will not be reduced unless \( b_1 \) is first reduced to \( \text{true} \).

Before we continue with the remaining small-step SOS rules for statements, let us see an example of a small-step SOS reduction using the rules discussed so far; as in the case of the big-step SOS rules in Section 3.2 recall that the small-step SOS rules are also rule schemas, that is, they are parametric in the (meta-)variables \( a, a_1, b, s, \sigma \), etc. The following is a correct derivation, where \( x \) and \( y \) are program variables and \( \sigma \) is any state with \( \sigma(x) = 1 \):

\[
\begin{align*}
\langle x, \sigma \rangle & \rightarrow \langle 1, \sigma \rangle \\
\langle y / x, \sigma \rangle & \rightarrow \langle y / 1, \sigma \rangle \\
\langle x + (y / x), \sigma \rangle & \rightarrow \langle x + (y / 1), \sigma \rangle \\
\langle (x + (y / x)) \leq x, \sigma \rangle & \rightarrow \langle (x + (y / 1)) \leq x, \sigma \rangle
\end{align*}
\]

The above can be regarded as a proof of the fact that replacing the second occurrence of \( x \) by 1 is a correct one-step computation of IMP, as defined using the small-step SOS rules discussed so far.

Let us now discuss the small-step SOS rules of statements in Figure 3.15. Unlike the reduction of expressions, a reduction step of a statement may also change the state. Note that the empty block, \( \{ \} \), acts as a value that statements evaluate to. Like for other values (integers and Booleans), there is no rule for the empty block, because such a rule would lead to non-termination. Rule (SMALLSTEP-BLOCK) simply dissolves the block construct and keeps the inner statement. We can afford to do this in IMP only because its blocks currently have no local variable declarations (this will change in IMP++ in Section 3.5). An alternative could be to keep the block and instead advance the inner statement (see Exercise 61). Rule (SMALLSTEP-ASGN-ARG2) reduces the second argument—which is an arithmetic expression—of an assignment statement whenever possible, regardless of whether the assigned variable was declared or not. Exercise 62 proposes an alternative semantics where the arithmetic expression is only reduced when the assigned variable has been declared. When the second argument is already fully reduced (i.e., it is an integer value), the rule (SMALLSTEP-ASGN) reduces the assignment statement to \( \{ \} \) at the same time updating the state accordingly. Therefore, two steps are needed in order to assign an already evaluated expression to a variable: one step to write the variable in the state and modify the assignment to \( \{ \} \), and another step to dissolve the resulting \( \{ \} \). Exercise 63 proposes an alternative semantics where these operations can be done in one step.

The rules (SMALLSTEP-SEQ-ARG1) and (SMALLSTEP-SEQ-EMPTY-BLOCK) give the small-step SOS of sequential composition: if the first statement is reducible then reduce it, otherwise, if it is \( \{ \} \), move on in a
\[
\langle s, \sigma \rangle \rightarrow \langle s, \sigma \rangle \quad \text{(SmallStep-Block)}
\]
\[
\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle \quad \text{(SmallStep-Assn-Arg2)}
\]
\[
\langle x = a; \sigma \rangle \rightarrow \langle x = a'; \sigma \rangle \quad \text{(SmallStep-Assn)}
\]
\[
\langle x = i; \sigma \rangle \rightarrow \langle [], \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \bot \quad \text{(SmallStep-Assn)}
\]
\[
\langle s_1, \sigma \rangle \rightarrow \langle s'_1, \sigma' \rangle \quad \text{(SmallStep-Seq-Arg1)}
\]
\[
\langle s_1, s_2, \sigma \rangle \rightarrow \langle s'_1, s'_2, \sigma' \rangle \quad \text{(SmallStep-Seq-Arg1)}
\]
\[
\langle \bot, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle \quad \text{(SmallStep-Seq-Empty-Block)}
\]
\[
\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle \quad \text{(SmallStep-If-Arg1)}
\]
\[
\langle \text{if (true) } s_1 \text{ else } s_2, \sigma \rangle \rightarrow \langle s_1, \sigma \rangle \quad \text{(SmallStep-If-True)}
\]
\[
\langle \text{if (false) } s_1 \text{ else } s_2, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle \quad \text{(SmallStep-If-False)}
\]
\[
\langle \text{while } (b) \ s, \sigma \rangle \rightarrow \langle \text{if } (b) \{ \ s \ \text{while } (b) \ s \} \text{ else } [], \sigma \rangle \quad \text{(SmallStep-While)}
\]
\[
\langle \text{int } x; \ s \rangle \rightarrow \langle s, x \mapsto 0 \rangle \quad \text{(SmallStep-Pgm)}
\]

Figure 3.15: SmallStep(IMP)—Small-step SOS of IMP statements \((i \in \text{Int}; x \in \text{Id}; xl \in \text{List} \{\text{Id}\}; a, a' \in AExp; b, b' \in BExp; s, s_1, s'_1, s_2 \in \text{Stmt}; \sigma, \sigma' \in \text{State})\).
Finally, configuration 

\[ C \rightarrow^* C \]  

(\text{SmallStep-Closure-Stop})

\[ C \rightarrow C', \quad C' \rightarrow^* C'' \quad \Rightarrow \quad C \rightarrow^* C'' \]  

(\text{SmallStep-Closure-More})

Figure 3.16: SmallStep(IMP)—Reflexive/transitive closure of the small-step SOS relation, which is the same for any small-step SOS of any language or calculus \((C, C', C'' \in \text{Configuration})\).

small-step to the second statement. Another possibility (different from that in Exercise 63) to avoid wasting the computational step generated by reductions to \(|\rangle|\) like in the paragraph above, is to eliminate the rule (SmallStep-Seq-Empty-Block) and instead to add a rule that allows the reduction of the second statement provided that the first one is terminated. This approach is proposed by Hennessy [82], where he introduces a new sequent for terminated configurations, say \(C \rightarrow^\ast\), and then includes a rule like the following (and no rule like (SmallStep-Seq-Empty-Block)):

\[ \langle s_1, \sigma \rangle \quad \quad \langle s_2, \sigma \rangle \rightarrow \langle s'_2, \sigma' \rangle \quad \Rightarrow \quad \langle s_1, s_2, \sigma \rangle \rightarrow \langle s'_2, \sigma' \rangle \]

The three rules for the conditional, namely (SmallStep-If-Arg1), (SmallStep-If-True), and (SmallStep-If-False), are straightforward; note that the two branches are never reduced when the condition can still be reduced. Exercise 65 proposes an alternative semantics for the conditional which wastes no computational step on switching to one of the two branches once the condition is evaluated.

The (SmallStep-While) rule unrolls the loop once; this unrolling semantics seems as natural as it can be, but one should notice that it actually also generates an artificial computational step. Exercise 66 proposes an alternative loop semantics which wastes no computational step.

Finally, (SmallStep-Pgm) gives the semantics of programs by reducing them to their body statement in the expected state formed by initializing all the declared variables to 0. Note, however, that this rule also wastes a computational step; indeed, one may not want the initialization of the state with default values for variables to count as a step. Exercise 67 addresses this problem.

It is worthwhile noting that one has some flexibility in how to give a small-step SOS semantics to a language. The same holds true for almost any language definitional style, not only for SOS.

On Proof Derivations, Evaluation, and Termination

To formally capture the notion of “sequence of transitions”, in Figure 3.16 we define the relation of reflexive/transitive closure of the small-step SOS transition.

\text{Definition 21.} Given appropriate IMP small-step SOS configurations \(C\) and \(C'\), the IMP small-step SOS sequent \(C \rightarrow C'\) is \textit{derivable}, written \(\text{SmallStep(IMP)} \vdash C \rightarrow C'\), iff there is some proof tree rooted in \(C \rightarrow C'\) which is derivable using the proof system \(\text{SmallStep(IMP)}\) in Figures 3.14 and 3.15. In this case, we also say that \(C\) \textit{reduces in one step} to \(C'\). Similarly, the many-step sequent \(C \rightarrow^* C'\) is \textit{derivable}, written \(\text{SmallStep(IMP)} \vdash C \rightarrow^* C'\), iff there is some proof tree rooted in \(C \rightarrow^* C'\) which is derivable using the proof system in Figures 3.14, 3.15, and 3.16. In this case, we also say that \(C\) \textit{reduces} (in zero, one, or more steps) to \(C'\). Configuration \(R\) is \textit{irreducible} iff there is no configuration \(C\) such that \(\text{SmallStep(IMP)} \vdash R \rightarrow C\), and is a \textit{result} iff it has one of the forms \(\langle i, \sigma \rangle\), \(\langle t, \sigma \rangle\), or \(\langle | \rangle, \sigma \rangle\), where \(i \in \text{Int}, t \in \text{Bool}, \text{and } \sigma \in \text{State}\). Finally, configuration \(C\) \textit{terminates} under \(\text{SmallStep(IMP)}\) iff there is no infinite sequence of configurations \(C_0, C_1, \ldots \) such that \(C_0 = C\) and \(C_i\) reduces in one step to \(C_{i+1}\) for any natural number \(i\).
Result configurations are irreducible, but there are irreducible configurations which are not necessarily result configurations. For example, the configuration \( \langle i / o, \sigma \rangle \) is irreducible but it is not a result. Like for big-step SOS, to catch division-by-zero within the semantics we need to add special error values/states and propagate them through all the language constructs (see Exercise 70).

The syntax of IMP (Section 3.1.1, Figure 3.1) was deliberately ambiguous with regards to sequential composition, and that was motivated by the fact that the semantics of the language will be given in such a way that the syntactic ambiguity will become irrelevant. We can now rigorously prove that is indeed the case, that is, we can prove properties of the like \( \text{SmallStep}(\text{IMP}) \vdash \langle (s_1 \ s_2) \ s_3, \sigma \rangle \rightarrow \langle (s'_1 \ s_2) \ s_3, \sigma' \rangle \) if and only if \( \text{SmallStep}(\text{IMP}) \vdash \langle s_1 \ s_2 \ s_3, \sigma \rangle \rightarrow \langle s'_1 \ s_2 \ s_3, \sigma' \rangle \)”, etc. Exercise 71 discusses several such properties which, together with the fact that the semantics of no language construct is structurally defined in terms of sequential composition, also says that adding the associativity of sequential composition as a structural identity to the small-step SOS of IMP does not change the set of global behaviors of any IMP program (although we have not added it). However, that will not be the case anymore when we extend IMP with dynamic threads in Section 3.5.4 because the semantics of thread spawning will be given making use of sequential composition in such a way that adding this structural identity will be necessary in order to capture the desired set of behaviors.

Note that there are non-terminating sequences which repeat configurations, as well as non-terminating sequences in which all the configurations are distinct; an example of the former is a sequence generated by reducing the statement \( \text{while} (\text{true}) \{ \} \) while an example of the latter is a sequence generated by reducing \( \text{while} (\! (n <\!= 0) \} \{ n = n + 1 ; \} \). Nevertheless, in the case of IMP, a configuration terminates if and only if it reduces to some irreducible configuration (see Exercise 72). This is not necessarily the case for non-deterministic languages, such as the IMP++ extension in Section 3.5 because reductions of configurations in such language semantics may non-deterministically choose steps that lead to termination, as well as steps that may not lead to termination. In the case of IMP though, the local non-determinism given by rules like \( \text{SmallStep-Add-Arg1} \) and \( \text{SmallStep-Add-Arg2} \) does not affect the overall determinism of the IMP language (Exercise 73).

Relating Big-Step and Small-Step SOS

As expected, the reflexive/transitive closure \( \rightarrow^* \) of the small-step SOS relation captures the same complete evaluation meaning as the \( \downarrow \) relation in big-step SOS (Section 3.2). Since for demonstrations reasons we deliberately worked with different result configurations in big-step and in small-step SOS, our theorem below looks slightly involved; if we had the same configurations in both semantics, then the theorem below would simply state “for any configuration \( C \) and any result configuration \( R \), \( \text{SmallStep}(\text{IMP}) \vdash C \rightarrow^* R \) if and only if \( \text{BigStep}(\text{IMP}) \vdash C \downarrow R \)”.

**Theorem 14.** The following equivalences hold for any \( a \in A\text{Exp}, \ i \in \text{Int}, b \in B\text{Exp}, t \in \text{Bool}, s \in \text{Stmt}, p \in \text{Pgm}, \sigma, \sigma' \in \text{State}:

- \( \text{SmallStep}(\text{IMP}) \vdash \langle a, \sigma \rangle \rightarrow^* \langle i, \sigma' \rangle \) for a state \( \sigma' \) iff \( \text{BigStep}(\text{IMP}) \vdash \langle a, \sigma \rangle \downarrow \langle i \rangle \);
- \( \text{SmallStep}(\text{IMP}) \vdash \langle b, \sigma \rangle \rightarrow^* \langle t, \sigma' \rangle \) for a state \( \sigma' \) iff \( \text{BigStep}(\text{IMP}) \vdash \langle b, \sigma \rangle \downarrow \langle t \rangle \);
- \( \text{SmallStep}(\text{IMP}) \vdash \langle s, \sigma \rangle \rightarrow^* \langle \{ \}, \sigma' \rangle \) for a state \( \sigma' \) iff \( \text{BigStep}(\text{IMP}) \vdash \langle s, \sigma \rangle \downarrow \langle \sigma' \rangle \);
- \( \text{SmallStep}(\text{IMP}) \vdash \langle p \rangle \rightarrow^* \langle \{ \}, \sigma \rangle \) for a state \( \sigma \) iff \( \text{BigStep}(\text{IMP}) \vdash \langle p \rangle \downarrow \langle \sigma \rangle \).

Note that the small-step SOS relation for IMP is a recursive, or decidable problem: indeed, given configurations \( C \) and \( C' \), one can use the small-step proof system in Figures 3.14 and 3.15 to exhaustively...
check whether indeed \( C \rightarrow C' \) is derivable or not. Moreover, since the rules for the reflexive/transitive closure relation \( \rightarrow^* \) in Figure 3.16 can be used to systematically generate any sequence of reductions, we conclude that the relation \( \rightarrow^* \) is \textit{recursively enumerable}. Theorem 1.4 together with the discussion at the end of Section 3.2.2 and Exercise 58 tell us that \( \rightarrow^* \) is properly recursively enumerable, that is, it cannot be recursive. This tells us, in particular, that non-termination of a program \( p \) is equivalent to saying that, no matter what the state \( \sigma \) is, \texttt{SmallStep}(IMP) \( \vdash \langle p \rangle \rightarrow^* \langle [\ell], \sigma \rangle \) cannot be derived. However, unlike for big-step SOS where nothing else can be said about non-terminating programs, in the case of small-step SOS definitions one can use the small-step relation, \( \rightarrow \), to observe program executions for any number of steps.

### 3.3.3 Small-Step SOS in Rewrite Logic

Like for big-step SOS, we can also associate a conditional rewrite rule to each small-step SOS rule and thereby obtain a rewrite logic theory that faithfully (i.e., step-for-step) captures the small-step SOS definition. Additionally, we can associate a rewrite logic equation to each SOS structural identity, because in both cases rules are applied modulo structural identities or equations. An important technical aspect needs to be resolved, though. The rewriting relation of rewrite logic is by its own nature reflexively and transitively closed. On the other hand, the small-step SOS relation is not reflexively and transitively closed by default (its reflexive/transitive closure is typically defined a posteriori, as we did in Figure 3.16). Therefore, we need to devise mechanisms to inhibit rewrite logic’s reflexive, transitive and uncontrolled application of rules.

We first show that any small-step SOS, say \texttt{SmallStep}, can be mechanically translated into a rewrite logic theory, say \( R_{\texttt{SmallStep}} \), in such a way that the corresponding derivation relations are step-for-step equivalent, that is, \texttt{SmallStep} \( \vdash C \rightarrow C' \) if and only if \( R_{\texttt{SmallStep}} \vdash \overrightarrow{R_{C \rightarrow C'}} \), where \( \overrightarrow{R_{C \rightarrow C'}} \) is the corresponding syntactic translation of the small-step SOS sequent \( C \rightarrow C' \) into a rewrite logic sequent. Second, we apply our generic translation technique on the small-step SOS formal system \texttt{SmallStep}(IMP) defined in Section 3.3.2 and obtain a rewrite logic semantics of IMP that is step-for-step equivalent to the original small-step SOS of IMP. Finally, we show how \( R_{\texttt{SmallStep}(IMP)} \) can be seamlessly defined in Maude, thus yielding another interpreter for IMP (in addition to the one similarly obtained from the big-step SOS of IMP in Section 3.2.3).

**Faithful Embedding of Small-Step SOS into Rewrite Logic**

Like for big-step SOS (Section 3.2.3), to define our translation from small-step SOS to rewrite logic generically, we assume that each parametric configuration \( C \) admits an equivalent algebraic variant \( \overrightarrow{C} \) as a term of sort \textit{Configuration} over an appropriate signature of configurations like the one that we defined for IMP in Section 3.2.1 (see Figure 3.6); moreover, each \textit{parameter} in \( C \) (e.g., arithmetic expression \( a \in AExp \)) gets replaced by a \textit{variable} of corresponding sort in \( \overrightarrow{C} \) (e.g., variable \( A \) of sort \( AExp \)). Consider now a general-purpose small-step SOS rule of the form

\[
\frac{C_1 \rightarrow C_1' \quad C_2 \rightarrow C_2' \quad \ldots \quad C_n \rightarrow C_n'}{C_0 \rightarrow C_0'} \quad \text{[if condition]}
\]

where \( C_0, C_1, C_1', C_2, C_2', \ldots, C_n, C_n' \) are configurations holding fragments of program together with all the needed semantic components, and \textit{condition} is an optional side condition. Before we introduce our transformation, let us first discuss why the same straightforward transformation that we used in the case of big-step SOS,

\[
(\forall \ell \forall \nu) \overrightarrow{C_0} \rightarrow \overrightarrow{C_0'} \text{ if } \overrightarrow{C_1} \rightarrow \overrightarrow{C_1'} \land \overrightarrow{C_2} \rightarrow \overrightarrow{C_2'} \land \ldots \land \overrightarrow{C_n} \rightarrow \overrightarrow{C_n'} \land \text{[\textit{condition}]},
\]

147
where $X$ is the set of parameters, or meta-variables, that occur in the small-step SOS rule, does not work in the case of small-step SOS. For example, with that transformation, the rewrite rules corresponding to the small-step SOS rules of IMP for assignment (SmallStep-Asgn-Arg2) and (SmallStep-Asgn) in Figure 3.15 would be

\[
\langle X = A ; , \sigma \rangle \rightarrow \langle X = A' ; , \sigma \rangle \quad \text{if} \quad \langle A , \sigma \rangle \rightarrow \langle A' , \sigma \rangle \\
\langle X = I ; , \sigma \rangle \rightarrow \langle \{ i \} , \sigma[I/X] \rangle \quad \text{if} \quad \sigma(X) \neq \perp
\]

The problem with these rules is that the rewrite of a configuration of the form $\langle x = i ; , \sigma \rangle$ for some $x \in Id$, $i \in Int$ and $\sigma \in State$ may not terminate, applying forever the first rule: in rewrite logic, $\langle i , \sigma \rangle \rightarrow \langle i , \sigma \rangle$ because $\rightarrow$ is closed under reflexivity. Even if we may somehow solve this reflexivity aspect by defining and then including an additional condition $A \neq A'$, such rules still fail to capture the intended small-step transition, because $\rightarrow$ is also closed transitively in rewrite logic, so there could be many small-steps taking place in the condition of the first rule before the rule is applied.

To capture exactly one step of reduction, thus avoiding the inherent automatic reflexive and transitive closure of the rewrite relation which is desirable in rewrite logic but not in reduction semantics, we can mark the left-hand-side (or, alternatively, the right-hand-side) configuration in each rewrite sequent to always be distinct from the other one; then each rewrite sequent comprises precisely one step, from a marked to an unmarked configuration (or vice versa). For example, let us place a $\circ$ in front of all the left-hand-side configurations and keep the right-hand-side configurations unchanged. Then the generic small-step SOS rule above translates into the rewrite logic rule

\[
(\forall X) \circ C_0 \rightarrow C_0 \quad \text{if} \quad \circ C_1 \rightarrow C_1 \land \circ C_2 \rightarrow C_2 \land \ldots \land \circ C_n \rightarrow C_n [\land \text{condition}],
\]

where $X$ is the same as above. One can metaphorically think of a marked configuration $\circ C$ as a “hot” configuration that needs to be “cooled down” in one step, while of an unmarked configuration $C$ as a cool one. Theorem 15 below states as expected that a small-step SOS sequent $C \rightarrow C'$ is derivable if and only if the term $\circ C$ rewrites in the corresponding rewrite theory to $C'$ (which is a normal form). Thus, to enable the resulting rewrite system on a given configuration, one needs to first mark the configuration to be reduced (by placing a $\circ$ in front of it) and then to let it rewrite to its normal form. Since the one-step reduction always terminates, the corresponding rewrite task also terminates.

If the original small-step SOS had structural identities, then we translate them into equations in a straightforward manner: each identity $\overline{t} = \overline{t}'$ is translated into an equation $(\forall X) \overline{t} = \overline{t}'$, where $X$ is the set of meta-variables appearing in the structural identity. The only difference between the original structural identity and the resulting equation is that the meta-variables of the former become variables in the latter. The role of the two is the same in their corresponding frameworks and whatever we can do with one in one framework we can equivalently do with the other in the other framework; consequently, to simplify the notation and the presentation, we will make abstraction of structural identities and equations in our theoretical developments in the remainder of this chapter.

To obtain the reflexive and transitive many-step closure of the small-step SOS relation in the resulting rewrite setting and thus to be able to obtain an interpreter for the defined language when executing the rewrite system, we need to devise some mechanism to iteratively apply the one-step reduction step captured by rewriting as explained above. There could be many ways to do that, but one simple and uniform way is to add a new configuration marker, say $\ast C$, with the meaning that $C$ must be iteratively reduced, small-step after small-step, either forever or until an irreducible configuration is reached. Figure 3.17 shows how one can define both configuration markers algebraically (assuming some existing Configuration sort, e.g., the one in Figure 3.13). To distinguish the marked configurations from the usual configurations and to also possibly allow several one-step markers at the same time, e.g., $\circ \circ \circ C$, which could be useful for debugging/tracing.

148
sorts:
    ExtendedConfiguration
subsorts:
    Configuration < ExtendedConfiguration
operations:
    ◦ : Configuration → ExtendedConfiguration // reduce one step
    ⋆ : Configuration → ExtendedConfiguration // reduce all steps
rule:
    ⋆Cfg → ⋆Cfg' if ◦ Cfg → Cfg' // where Cfg, Cfg' are variables of sort Configuration

Figure 3.17: Representing small-step and many-step SOS reductions in rewrite logic.

reasons, we preferred to define the sort of marked configurations as a supersort of Configuration. Note that the rule in Figure 3.17 indeed gives ⋆ its desired reflexive and transitive closure property (the reflexivity follows from the fact that the rewrite relation in rewrite logic is reflexive, so ⋆C → ⋆C for any configuration term C).

Theorem 15. (Faithful embedding of small-step SOS into rewrite logic) For any small-step SOS SmallStep, and any SmallStep appropriate configurations C and C', the following equivalences hold:

\[ \text{SmallStep} ⊢ C → C' ⇐⇒ \text{R}_{\text{SmallStep}} ⊢ ◦ \overline{C} → \overline{C'} \]

\[ \text{SmallStep} ⊢ C → ⋆ C' ⇐⇒ \text{R}_{\text{SmallStep}} ⊢ ⋆ \overline{C} → ⋆ \overline{C'} \]

where \( \text{R}_{\text{SmallStep}} \) is the rewrite logic semantic definition obtained from SmallStep by translating each rule in SmallStep as above. (Recall from Section 2.5 that →₁ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewrite logic. Also, as C and C' are parameter-free—parameters only appear in rules—, \( \overline{C} \) and \( \overline{C'} \) are ground terms.)

Except for transforming parameters into variables, the only apparent difference between SmallStep and \( \text{R}_{\text{SmallStep}} \) is that the latter marks (using ◦) all the left-hand-side configurations and, naturally, uses conditional rewrite rules instead of conditional deduction rules. As Theorem 15 shows, there is a step-for-step correspondence between their corresponding computations (or executions, or derivations). Therefore, similarly to the big-step SOS representation in rewrite logic, the rewrite theory \( \text{R}_{\text{SmallStep}} \) is the small-step SOS SmallStep, and not an encoding of it.

Recall from Section 3.2.3 that in the case of big-step SOS there were some subtle differences between the one-step \( →₁ \) (obtained by dropping the reflexivity and transitivity rules of rewrite logic) and the usual \( → \) relations in the rewrite theory corresponding to the big-step SOS. The approach followed in this section based on marking configurations, thus keeping the left-hand and the right-hand-sides always distinct, eliminates all the differences between the two rewrite relations in the case of the one-step reduction (the two relations are identical on the terms of interest). The second equivalence in Theorem 15 tells us that we can turn the rewrite logic representation of the small-step SOS language definition into an interpreter by simply marking the configuration to be completely reduced with a ⋆ and then letting the rewrite engine do its job.

It is worthwhile noting that like in the case of the big-step SOS representation in rewrite logic, unfortunately, \( \text{R}_{\text{SmallStep}} \) lacks the main strengths of rewrite logic: in rewrite logic, rewrite rules can apply under any context and in parallel. Indeed, the rules of \( \text{R}_{\text{SmallStep}} \) can only apply at the top, sequentially. This should not surprise because, as stated, \( \text{R}_{\text{SmallStep}} \) is SmallStep, with all its strengths and limitations. By all means,
both the $R_{\text{SmallStep}}$ above and the $R_{\text{BigStep}}$ in Section 3.2.3 are rather poor-style rewrite logic specifications. However, that is normal, because neither big-step SOS nor small-step SOS were meant to have the capabilities of rewrite logic w.r.t. context-insensitivity and parallelism; since their representations in rewrite logic are faithful, one should not expect that they inherit the additional capabilities of rewriting logic (if they did, then the representations would not be step-for-step faithful, so something would be wrong).

Small-Step SOS of IMP in Rewrite Logic

Figure 3.18 gives the rewrite logic theory $R_{\text{SmallStep}}(\text{IMP})$ that is obtained by applying the procedure above to the small-step SOS of IMP, namely the formal system $\text{SmallStep}(\text{IMP})$ presented in Figures 3.14 and 3.15. As usual, we used the rewrite logic convention that variables start with upper-case letters, and like in the rewrite theory corresponding to the big-step SOS of IMP in Figure 3.8, we used $\sigma$ (a larger $\sigma$ symbol) for variables of sort $\text{State}$. Besides the parameter vs. variable subtle (but not unexpected) aspect, the only perceivable difference between $\text{SmallStep}(\text{IMP})$ and $R_{\text{SmallStep}}(\text{IMP})$ is the different notational conventions they use. The following corollary of Theorem 15 establishes the faithfulness of the representation of the small-step SOS of IMP in rewriting logic:

**Corollary 5.** $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow C' \iff R_{\text{SmallStep}}(\text{IMP}) \vdash \sigma C \rightarrow \sigma C'$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system $\text{SmallStep}(\text{IMP})$ and generic rewrite logic deduction using the IMP-specific rewrite rules in $R_{\text{SmallStep}}(\text{IMP})$; the two are faithfully equivalent.

★ Maude Definition of IMP Small-Step SOS

Figure 3.19 shows a straightforward Maude representation of the rewrite theory $R_{\text{SmallStep}}(\text{IMP})$ in Figure 3.18 including representations of the algebraic signatures of small-step SOS configurations in Figure 3.13 and of their extensions in Figure 3.17, which are needed to capture small-step SOS in rewrite logic. The Maude module $\text{IMP-SEMANTICS-SMALLSTEP}$ in Figure 3.19 is executable, so Maude, through its rewriting capabilities, yields a small-step SOS interpreter for IMP the same way it yielded a big-step SOS interpreter in Section 3.2.3; for example, the command

```
rewrite * < sumPgm > .
```

where `sumPgm` is the first program defined in the module $\text{IMP-PROGRAMS}$ in Figure 3.4 produces a result of the form (the exact statistics are also irrelevant, so they were replaced by “…”):

```
rewrites: 7632 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < {},n |-> 0 & s |-> 5050 >
```

Like for the big-step SOS definition in Maude, one can also use any of the general-purpose tools provided by Maude on the small-step SOS definition above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. However, a relatively large number of states will be explored, 1709, due to the non-deterministic evaluation strategy of the various language constructs:

```
Solution 1 (state 1708)
states: 1709 rewrites: 9232 in ... cpu (... real) (... rewrites/second)
Cfg:ExtendedConfiguration --> * < {},n |-> 0 & s |-> 5050 >
```
\[\circ(X, \sigma) \rightarrow \langle \sigma(X), \sigma' \rangle \text{ if } \sigma(X) \neq \bot\]

\[\circ(A_1 + A_2, \sigma) \rightarrow \langle A'_1 + A_2, \sigma' \rangle \text{ if } \circ(A_1, \sigma) \rightarrow \langle A'_1, \sigma' \rangle\]

\[\circ(A_1 + A_2, \sigma) \rightarrow \langle A_1 + A'_2, \sigma' \rangle \text{ if } \circ(A_2, \sigma) \rightarrow \langle A'_2, \sigma' \rangle\]

\[\circ(I_1 + I_2, \sigma) \rightarrow \langle I_1 +_{\sigma} I_2, \sigma' \rangle\]

\[\circ(A_1 / A_2, \sigma) \rightarrow \langle A'_1 / A_2, \sigma' \rangle \text{ if } \circ(A_1, \sigma) \rightarrow \langle A'_1, \sigma' \rangle\]

\[\circ(A_1 / A_2, \sigma) \rightarrow \langle A_1 / A'_2, \sigma' \rangle \text{ if } \circ(A_2, \sigma) \rightarrow \langle A'_2, \sigma' \rangle\]

\[\circ(I_1 / I_2, \sigma) \rightarrow \langle I_1 /_{\sigma} I_2, \sigma' \rangle \text{ if } I_2 \neq 0\]

\[\circ(A_1 \ll A_2, \sigma) \rightarrow \langle A'_1 \ll A_2, \sigma' \rangle \text{ if } \circ(A_1, \sigma) \rightarrow \langle A'_1, \sigma' \rangle\]

\[\circ(I_1 \ll A_2, \sigma) \rightarrow \langle I_1 \ll A'_2, \sigma' \rangle \text{ if } \circ(A_2, \sigma) \rightarrow \langle A'_2, \sigma' \rangle\]

\[\circ(I_1 \ll I_2, \sigma) \rightarrow \langle I_1 \ll_{\sigma} I_2, \sigma' \rangle\]

\[\circ(! B, \sigma) \rightarrow \langle ! B', \sigma' \rangle \text{ if } \circ(B, \sigma) \rightarrow \langle B', \sigma' \rangle\]

\[\circ(! \text{ true}, \sigma) \rightarrow \langle \text{false}, \sigma' \rangle\]

\[\circ(! \text{ false}, \sigma) \rightarrow \langle \text{true}, \sigma' \rangle\]

\[\circ(B_1 \& \& B_2, \sigma) \rightarrow \langle B'_1 \& \& B_2, \sigma' \rangle \text{ if } \circ(B_1, \sigma) \rightarrow \langle B'_1, \sigma' \rangle\]

\[\circ(\text{false} \& \& B_2, \sigma) \rightarrow \langle \text{false}, \sigma' \rangle\]

\[\circ(\text{true} \& \& B_2, \sigma) \rightarrow \langle B_2, \sigma' \rangle\]

\[\circ([S], \sigma) \rightarrow \langle S, \sigma' \rangle\]

\[\circ(X = A; \sigma) \rightarrow \langle X = A'; \sigma' \rangle \text{ if } \circ(A, \sigma) \rightarrow \langle A', \sigma' \rangle\]

\[\circ(X = I; \sigma) \rightarrow \langle [], \sigma[I/X] \rangle \text{ if } \sigma(X) \neq \bot\]

\[\circ(S_1 S_2, \sigma) \rightarrow \langle S'_1 S_2, \sigma' \rangle \text{ if } \circ(S_1, \sigma) \rightarrow \langle S'_1, \sigma' \rangle\]

\[\circ([], S_2, \sigma) \rightarrow \langle S_2, \sigma' \rangle\]

\[\circ(\text{if} (B) S_1 \text{ else } S_2, \sigma) \rightarrow \langle \text{if} (B') S_1 \text{ else } S_2, \sigma' \rangle \text{ if } \circ(B, \sigma) \rightarrow \langle B', \sigma' \rangle\]

\[\circ(\text{if} (\text{true}) S_1 \text{ else } S_2, \sigma) \rightarrow \langle S_1, \sigma' \rangle\]

\[\circ(\text{if} (\text{false}) S_1 \text{ else } S_2, \sigma) \rightarrow \langle S_2, \sigma' \rangle\]

\[\circ(\text{while} (B) S, \sigma) \rightarrow \langle \text{if} (B) [S \text{ while } (B) S] \text{ else } [], \sigma' \rangle\]

\[\circ(\text{int } X; S) \rightarrow \langle S, (X \mapsto 0) \rangle\]

Figure 3.18: \(R_{\text{SMALLEST}}(\text{IMP})\): the small-step SOS of IMP in rewrite logic.
mod IMP-CONFIGURATIONS-SMALLSTEP is including IMP-SYNTAX + STATE .
sorts Configuration ExtendedConfiguration .
subsort Configuration < ExtendedConfiguration .
op <_,_> : AExp State -> Configuration .
op <_,_> : BExp State -> Configuration .
op <_,_> : Stmt State -> Configuration .
op <_> : Pgm -> Configuration .
op o_ : Configuration -> ExtendedConfiguration [prec 80] . --- one step
op o_ : Configuration -> ExtendedConfiguration [prec 80] . --- all steps
var Cfg Cfg' : Configuration .
crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SEMANTICS-SMALLSTEP is including IMP-CONFIGURATIONS-SMALLSTEP .
var X : Id . var Sigma Sigma' : State . var I I1 I2 : Int . var Xl : List{Id} .
var A A' A1 A2 A1' A2' : AExp . var B B' B1 B1' B2 : BExp . var S S' S1 S1' S2 : Stmt .
crl o < X,Sigma > => < Sigma(X),Sigma > if Sigma(X) /=/ Bool undefined .
crl o < A1 + A2,Sigma > => < A1' + A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
crl o < A1 + A2,Sigma > => < A1 + A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
rl o < I1 + I2,Sigma > => < I1 +Int I2,Sigma > .
crl o < A1 / A2,Sigma > => < A1' / A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
crl o < A1 / A2,Sigma > => < A1 / A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
crl o < A1 / A2,Sigma > => < A1 / A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
rl o < I1 / I2,Sigma > => < I1 /Int I2,Sigma > if I2 /=/ Bool 0 .
crl o < A1 <= A2,Sigma > => < A1' <= A2,Sigma > if o < A2,Sigma > => < A2',Sigma > .
crl o < I1 <= A2,Sigma > => < I1 <= A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
rl o < I1 <= I2,Sigma > => < I1 <=Int I2,Sigma > .
crl o < ! B,Sigma > => < ! B',Sigma > if o < B,Sigma > => < B',Sigma > .
rl o < ! true,Sigma > => < false,Sigma > .
rl o < ! false,Sigma > => < true,Sigma > .
crl o < B1 && B2,Sigma > => < B1' && B2,Sigma > if o < B1,Sigma > => < B1',Sigma > .
rl o < false && B2,Sigma > => < false,Sigma > .
rl o < true && B2,Sigma > => < false,Sigma > .
rl o < {S},Sigma > => < S,Sigma > .
crl o < X = A ;,Sigma > => < X = A' ;,Sigma > if o < A,Sigma > => < A',Sigma > .
crl o < X = I ;,Sigma > => < {},Sigma[I / X] > if Sigma(X) /=/ Bool undefined .
crl o < S1 S2,Sigma > => < S1' S2,Sigma' > if o < S1,Sigma > => < S1',Sigma' > .
rl o < [{}],S2,Sigma > => < S2,Sigma > .
crl o < if (B) S1 else S2,Sigma > => < if (B') S1 else S2,Sigma > if o < B,Sigma > => < B',Sigma > .
rl o < if (true) S1 else S2,Sigma > => < S1,Sigma > .
rl o < if (false) S1 else S2,Sigma > => < S2,Sigma > .
rl o < while (B) S,Sigma > => < if (B) {S while (B) S} else {},Sigma > .
rl o < int X1 ; S > => < S,(X1 |-> 0) > .
endm

Figure 3.19: The small-step SOS of IMP in Maude, including the definition of configurations.

152
3.3.4 Notes

Small-step structural operational semantics was introduced as just structural operational semantics (SOS; no “small-step” qualifier at that time) by Plotkin in a 1981 technical report (University of Aarhus Technical Report DAIMI FN-19, 1981) that included his lecture notes of a programming language course [60]. For more than 20 years this technical report was cited as the main SOS reference by hundreds of scientists who were looking for mathematical rigor in their programming language research. It was only in 2004 that Plotkin’s SOS was finally published in a journal [61].

Small-step SOS is pedagogically discussed in several textbooks, two early notable ones being Hennessy [32] (1990) and Winskel [87] (1993). Hennessy [32] uses the same notation as Plotkin, but Winskel [87] prefers a different one to make it clear that it is a one step semantics: \( \langle s, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle \). Like for big-step SOS, there is no well-established notation for small-step SOS sequents. There is a plethora of research projects and papers that explicitly or implicitly take SOS as the formal language semantics framework. Also, SOS served as a source of inspiration, or of problems to be fixed, to other semantic framework designers, including the author. There is simply too much work on SOS, using it, or modifying it, to attempt to cover it here. We limit ourselves to directly related research focused on capturing SOS as a methodological fragment of rewrite logic.

The marked configuration style that we adopted in this section to faithfully represent small-step SOS in rewrite logic was borrowed from Ţerăbanută et al. [74]; there, the configuration marker “\( \sigma \)” was called a “configuration modifier”. An alternative way to keep the left-hand and the right-hand-side configurations distinct was proposed by Meseguer and Braga in [44, 14] in the context of representing MSOS into rewrite logic (see Section 3.6); the idea there was to use two different types of configuration wrappers, one for the left-hand-side of the transitions and one for the right-hand-side, yielding rewrite logic rules of the form:

\[
(\forall \lambda) [C_0] \rightarrow [C'_0] \quad \text{if} \quad [C_1] \rightarrow [C'_1] \land [C_2] \rightarrow [C'_2] \land \cdots \land [C_n] \rightarrow [C'_n] \land \text{condition}.
\]

The solution proposed by Meseguer and Braga in [44, 14] builds upon experience with a previous representation of MSOS in rewrite logic in [15] as well as with an implementation of it in Maude [13, 16], where the necessity of being able to inhibit the default reflexivity and transitivity of the rewrite relation took shape. We preferred to follow the configuration modifier approach proposed by Ţerăbanută et al. [74] because it appears to be slightly less intrusive (we only tag the already existing left-hand terms of rules) and more general (the left-hands of rules can have any structure, not only configurations, including no structure at all, as it happens in most of the rules of reduction semantics with evaluation contexts—see Section 3.7, e.g., Figure 3.40).

Vardejo and Martí-Oliet [84] give a Maude implementation of a small-step SOS definition for a simple imperative language similar to our IMP (Hennessy’s WhileL language [32]), in which they do not attempt to prevent the inherent transitivity of rewriting. While they indeed obtain an executable semantics that is reminiscent of the original small-step SOS of the language, they actually define directly the transitive closure of the small-step SOS relation; they explicitly disable the reflexive closure by checking \( C \neq C' \) next to rewrites \( C \rightarrow C' \) in rule conditions. A small-step SOS of a simple functional language (Hennessy’s Fpl language [32]) is also given in [84], following a slightly different style, which avoids the problem above. They successfully inhibit rewriting’s inherent transitivity in their definition by using a rather creative rewriting representation style for sequents. More precisely, they work with sequents which appear to the user as having the form \( \sigma \vdash a \rightarrow a' \), where \( \sigma \) is a state and \( a, a' \) are arithmetic expressions, etc., but they actually are rewrite relations between terms \( \sigma \vdash a \) and \( a' \) (an appropriate signature to allow that to parse is defined). Indeed, there is no problem with the automatic reflexive/transitive closure of rewriting here because the LHS and the RHS of each rewrite rule have different structures. The simple functional language in [84] was pure (no side effects), so there was no need to include a resulting state in their rules; if the language had
side effects, then this Vardejo and Marti-Oliet’s representation of small-step SOS sequents in [84] would effectively be the same as the one by Meseguer and Braga in [44, 14] (but using a different notation).

### 3.3.5 Exercises

Prove the following exercises, all referring to the IMP small-step SOS in Figures [3.14] and [3.15]

**Exercise 59.** Change the small-step rules for `/` so that it short-circuits when $a_1$ evaluates to 0.

**Exercise 60.** Change the small-step SOS of the IMP conjunction so that it is not short-circuited.

**Exercise 61.** Change the small-step SOS of blocks so that the block is kept but its inner statement is advanced one step.

**Exercise 62.** One can rightfully argue that the arithmetic expression in an assignment should not be reduced any step when the assigned variable is not declared. Change the small-step SOS of IMP to only reduce the arithmetic expression when the assigned variable is declared.

**Exercise 63.** A sophisticated language designer could argue that the reduction of the assignment statement to emptyBlockIMP is an artifact of using small-step SOS, therefore an artificial and undesired step which affects the intended computational granularity of the language. Change the small-step SOS of IMP to eliminate this additional small-step.

*Hint: Follow the style in Exercise 68; note, however, that that style will require more rules and more types of configurations, so from that point of view is more complex.*

**Exercise 64.** Give a proof system for deriving “terminated configuration” sequents $C \sqrt{.}$

**Exercise 65.** One could argue that our small-step SOS rules for the conditional waste a computational step when switching to one of the two branches once the condition is evaluated.

1. Give an alternative small-step SOS for the conditional which does not require a computational step to switch to one of the two branches.

2. Can one do better than that? That is, can one save an additional step by reducing the corresponding branch one step at the same time with reducing the condition to true or false in one step? *Hint: one may need terminated configurations, like in Exercise 64*.

**Exercise 66.** Give an alternative small-step SOS definition of while loops which wastes no computational step. *Hint: do a case analysis on b, like in the rules for the conditional.*

**Exercise 67.** Give an alternative small-step SOS definition of variable declarations which wastes no computational steps. *Hint: one may need terminated configurations, like in Exercise 64.*

**Exercise 68.** Modify the small-step SOS definition of IMP such that the configurations in the right-hand-sides of the transition sequents are minimal (they should contain both a fragment of program and a state only when absolutely needed). What are the drawbacks of this minimalistic approach, compared to the small-step SOS semantics that we chose to follow?

**Exercise 69.** Show that the small-step SOS resulting from Exercise 68 is equivalent to the one in Figure [3.14] on arithmetic and Boolean expressions, that is, $\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle$ is derivable with the proof system in Figure [3.14] if and only if $\langle a, \sigma \rangle \rightarrow \langle a' \rangle$ is derivable with the proof system in Exercise 68 and similarly for Boolean expressions. However, show that the equivalence does not hold true for statements.
Exercise 70. To handle division-by-zero, add “error” values and statements, and modify the small-step SOS in Figures 3.14 and 3.15 to allow derivations of sequents whose right-hand-side configurations contain “error” as their syntactic component. See also Exercise 56 (same problem but for big-step SOS).

Exercise 71. For any IMP statements $s_1, s'_1, s_2, s_3$ and any states $\sigma, \sigma'$, the following hold:

1. $\text{SmallStep}(\text{IMP}) \vdash \langle \{\} s_2 s_3, \sigma \rangle \rightarrow \langle s_2 s_3, \sigma \rangle$ and
   $\text{SmallStep}(\text{IMP}) \vdash \langle \{\} (s_2 s_3), \sigma' \rangle \rightarrow \langle s_2 s_3, \sigma' \rangle$; and

2. $\text{SmallStep}(\text{IMP}) \vdash \langle s_1 (s_2 s_3), \sigma \rangle \rightarrow \langle s'_1 (s_2 s_3), \sigma' \rangle$ if and only if
   $\text{SmallStep}(\text{IMP}) \vdash \langle s_1 s_2 s_3, \sigma \rangle \rightarrow \langle s'_1 s_2 s_3, \sigma' \rangle$.

Consequently, the following also hold (prove them by structural induction on $s_1$):

$\text{SmallStep}(\text{IMP}) \vdash \langle (s_1 s_2) s_3, \sigma \rangle \rightarrow^* \langle s_2 s_3, \sigma' \rangle$ if and only if
$\text{SmallStep}(\text{IMP}) \vdash \langle s_1 (s_2 s_3), \sigma \rangle \rightarrow^* \langle s'_1 s_2 s_3, \sigma' \rangle$ if and only if
$\text{SmallStep}(\text{IMP}) \vdash \langle s_1, \sigma \rangle \rightarrow^* \langle \{\}, \sigma' \rangle$.

Exercise 72. With the SmallStep(IMP) proof system in Figures 3.14, 3.15, and 3.16, configuration $C$ terminates if $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow^* R$ for some irreducible configuration $R$.

Exercise 73. The small-step SOS of IMP is globally deterministic: if $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow^* R$ and $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow^* R'$ for irreducible configurations $R$ and $R'$, then $R = R'$. Show the same result for the proof system detecting division-by-zero as in Exercise 70.

Exercise 74. Show that if $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow^* \langle i, \sigma \rangle$ for some configuration $C$, integer $i$, and state $\sigma$, then $C$ must be of the form $\langle a, \sigma \rangle$ for some arithmetic expression $a$. Show a similar result for Boolean expressions. For statements, show that if $\text{SmallStep}(\text{IMP}) \vdash C \rightarrow^* \langle \{\}, \sigma \rangle$ then $C$ must be either of the form $\langle s, \sigma' \rangle$ for some statement $s$ and some state $\sigma'$, or of the form $\langle p \rangle$ for some program $p$.

Exercise 75. Prove Theorem 74.

Exercise 76. State and show a result similar to Theorem 74 but for the small-step and big-step SOS proof systems in Exercises 70 and 56 respectively.
Is this enough? Should I add more citations here? Ask Narciso, Peter and Jose for their and others' works embedding small-step into rewrite logic.

add the variables in rewrite rules
3.4 Denotational Semantics

Denotational semantics, also known as fixed-point semantics, associates to each syntactically well-defined fragment of program a well-defined, rigorous mathematical object. This mathematical object denotes the complete behavior of the fragment of program, no matter in what context it will be used. In other words, the denotation of a fragment of program represents its contribution to the meaning of any program containing it. In particular, equivalence of programs or fragments is immediately translated into equivalence of mathematical objects. The later can be then shown using the entire arsenal of mathematics, which is supposedly better understood and more well-established than that of the relatively much newer field of programming languages. There are no theoretical requirements on the nature of the mathematical domains in which the fragments of program are interpreted, although a particular approach became quite well-established, to an extent that it is by many identified with denotational semantics itself: choose the domains to be appropriate bottomed complete partial orders (abbreviated CPOs; see Section 2.6), and give the denotation of recursive language constructs (including loops, recursive functions, recursive data-structures, recursive types, etc.) as least fixed-points, which exist thanks to Theorem 12.

Each language requires customized denotational semantics, the same way each language required customized big-step or small-step structural operational semantics in the previous sections in this chapter. For the sake of concreteness, below we discuss general denotational semantics notions and notations by means of our running example language, IMP, without attempting to completely define it. Note that the complete denotational semantics of IMP is listed in Figure 3.20 in Section 3.4.1. Consider, for example, arithmetic expressions in IMP (which are side-effect free). Each arithmetic expression can be thought of as the mathematical object which is a partial function taking a state to an integer value, namely the value that the expression evaluates to in the given state. It is a partial function and not a total one because the evaluation of some arithmetic expressions may not be defined in some states, for example due to illegal operations such as division by zero. Thus, we can define the denotation of arithmetic expressions as a total function

\[ [\cdot] : AExp \to (State \to Int) \]

taking arithmetic expressions to partial functions from states to integer numbers. As in all the semantics discussed in this chapter, states are themselves partial maps from names to values. In what follows we will follow a common notational simplification and will write \([a]_\sigma\) instead of \([a]_\sigma(\sigma)\) whenever \(a \in AExp\) and \(\sigma \in State\), and similarly for other syntactic or semantic categories. To avoid ambiguity in the presence of multiple denotation functions, many works on denotational semantics tag the denotation functions with their corresponding syntactic categories, e.g., \(AExp[\cdot]\) or \([\cdot]_{AExp}\). Our IMP language is simple enough that we can afford to not add such tags.

Denotation functions are defined inductively, over the structure of the language constructs. For example, if \(i \in In\) then \([i]_\sigma\) is the constant function \(i\), that is, \([i]_\sigma = i\) for any \(\sigma \in State\). Similarly, if \(x \in Id\) then \([x]_\sigma = \sigma(x)\). As it is the case in mathematics, if an undefined value is used to calculate another value, then the resulting value is also undefined. In particular, \([x]_\sigma\) is undefined when \(x \notin \text{Dom}(\sigma)\). The denotation of compound constructs is defined in terms of the denotations of the parts. In other words, we say that denotational semantics is compositional. For example, \([a_1 + a_2]_\sigma = [a_1]_\sigma +_{\text{Int}} [a_2]_\sigma\) for any \(a_1, a_2 \in AExp\) and any \(\sigma \in State\). For the same reason as above, if any of \([a_1]_\sigma\) or \([a_2]_\sigma\) is undefined then \([a_1 + a_2]_\sigma\) is also implicitly undefined. One can also chose to explicitly keep certain functions undefined in certain states, such as the denotation of division in those states in which the denominator is zero:

\[
[a_1 / a_2]_\sigma = \begin{cases} 
[a_1]_\sigma /_{\text{Int}} [a_2]_\sigma & \text{if } [a_2]_\sigma \neq 0 \\
\perp & \text{if } [a_2]_\sigma = 0
\end{cases}
\]
Note that even though the case where \([a_2] / \sigma\) is undefined (⊥) was not needed to be explicitly listed above (because it falls under the first case, \([a_2] / \sigma \neq 0\)), it is still the case that \([a_1 / a_2] \sigma\) is undefined whenever any of \([a_1] \sigma\) or \([a_2] \sigma\) is undefined.

An immediate use of denotational semantics is to prove properties about programs. For example, we can show that the addition operation on \(AExp\) whose denotational semantics was given above is associative. Indeed, we can prove for any \(a_1, a_2, a_3 \in AExp\) the following equality of partial functions

\[
(a_1 + a_2) + a_3 = a_1 + (a_2 + a_3)
\]

using conventional mathematical reasoning and the fact that the sum \(+_\text{Int}\) in the \(\text{Int}\) domain is associative (see Exercise 77). Note that denotational semantics allows us not only to prove properties about programs or fragments of programs relying on properties of their mathematical domains of interpretation, but also, perhaps even more importantly, it allows us to elegantly formulate such properties. Indeed, what does it mean for a language construct to be associative or, in general, for any desired property over programs or fragments of programs to hold? While one could use any of the operational semantics discussed in this chapter to answer this, denotational semantics gives us one of the most direct means to state and prove program properties.

Each syntactic category is interpreted into its corresponding mathematical domain. For example, the denotations of Boolean expressions and of statements are total functions of the form:

- \([\_] : BExp \rightarrow (\text{State} \rightarrow \text{Bool})\)
- \([\_] : \text{Stmt} \rightarrow (\text{State} \rightarrow \text{State})\)

The former is similar to the one for arithmetic expressions above, so we do not discuss it here. The latter is more interesting. Statements can indeed be regarded as partial functions taking states into resulting states. In addition to partiality due to illegal operations in expressions that statements may involve, such as division by zero, partiality in the denotation of statements may also occur for another important reason: loops may not terminate. For example, the statement \(\text{while} (x <= y) \{\}\) will not terminate in those states in which the value that \(x\) denotes is less than or equal to that of \(y\). Mathematically, we say that the function from states to states that this loop statement denotes is undefined in those states in which the loop statement does not terminate. This will be elaborated shortly, after we discuss other statement constructs.

Since \(\{\}\) does not change the state, its denotation is the identity function, i.e., \([\{}\] = \(1_{\text{State}}\). The assignment statement updates the given state when defined in the assigned variable, that is, \([x = a;] \sigma \} = \sigma[a/\sigma/x]\}

when \(\sigma(x) \neq \perp\) and \([a] \sigma \neq \perp\), and \([x = a;] \sigma = \perp\) otherwise. Sequential composition accumulates the state changes of the denotations of the composed statements, so it is precisely the mathematical composition of the corresponding partial functions: \([s_1; s_2] = [s_2] \circ [s_1]\].

As an example, let us calculate the denotation of the statement \("x = 1; y = 2; x = 3;\"\) when \(x \neq y\), i.e., the function \([x = 1; y = 2; x = 3;]\}. Applying the denotation of sequential composition twice, we obtain \([x = 3;] \circ [y = 2;] \circ [x = 1;]\}. Applying this composed function on a state \(\sigma\}, one gets \([x = 3;] \circ [y = 2;] \circ [x = 1;]\} \sigma = \sigma[1/x][2/y][3/x]\} when \(\sigma(x)\) and \(\sigma(y)\) are both defined, and equals \(\perp\) when any of \(\sigma(x)\) or \(\sigma(y)\) is undefined; let \(\sigma'\) denote \(\sigma[1/x][2/y][3/x]\}. By the definition of function update, one can easily see that \(\sigma'\) can be defined as

\[
\sigma'(z) = \begin{cases} 
3 & \text{if } z = x \\
2 & \text{if } z = y \\
\sigma(z) & \text{otherwise,}
\end{cases}
\]

which is nothing but \(\sigma[2/y][3/x]\}. We can therefore conclude that the statements \("x = 1; y = 2; x = 3;\"\) and \("y = 2; x = 3;\"") are equivalent, because they have the same denotation.
The denotation of a conditional statement if \((b)\) \(s_1\) else \(s_2\) in a state \(\sigma\) is either the denotation of \(s_1\) in \(\sigma\) or that of \(s_2\) in \(\sigma\), depending upon the denotation of \(b\) in \(\sigma\):

\[
\llbracket\text{if}(b)\text{else} s_2\rrbracket\sigma = \begin{cases} \llbracket s_1\rrbracket\sigma & \text{if } \llbracket b\rrbracket\sigma = \text{true} \\ \llbracket s_2\rrbracket\sigma & \text{if } \llbracket b\rrbracket\sigma = \text{false} \\ \bot & \text{if } \llbracket b\rrbracket\sigma = \bot \end{cases}
\]

The third case above was necessary, because the first two cases do not cover the entire space of possibilities and, in such situations, one may (wrongly in our context here) understand that the function is underspecified in the remaining cases rather than undefined. Using the denotation of the conditional statement above and conventional mathematical reasoning, we can show, for example, that \(\llbracket\text{if}(y \leq z)\{x = 1;\}\text{else}\{x = 2;\}\}x = 3;\) is the function taking states \(\sigma\) defined in \(x\), \(y\) and \(z\) to \(\sigma[3/x]\).

The language constructs which admit non-trivial and interesting denotational semantics tend to be those which have a recursive nature. One of the simplest such constructs, and the only one we discuss here (see Section 4.7 for other recursive constructs), is IMP’s while looping construct. Thus, the question we address next is how to define the denotation functions of the form

\[
\llbracket\text{while}(b)\text{else} s\rrbracket : \text{State} \rightarrow \text{State}
\]

where \(b \in BExp\) and \(s \in Stmt\). What we want is \(\llbracket \text{while}(b)\text{else} s\rrbracket\sigma = \sigma'\) iff the while loop correctly terminates in state \(\sigma'\) when executed in state \(\sigma\). Such a \(\sigma'\) may not always exist for two reasons:

1. Because \(b\) or \(s\) is undefined (e.g., due to illegal operations) in \(\sigma\) or in other states encountered during the loop execution; or
2. Because \(s\) (which may contain nested loops) or the while loop itself does not terminate.

If \(w\) is the partial function \(\llbracket\text{while}(b)\text{else} s\rrbracket\), then its most natural definition would appear to be:

\[
w(\sigma) = \begin{cases} w(\llbracket s\rrbracket\sigma) & \text{if } \llbracket b\rrbracket\sigma = \text{true} \\ \sigma & \text{if } \llbracket b\rrbracket\sigma = \text{false} \\ \bot & \text{if } \llbracket b\rrbracket\sigma = \bot \end{cases}
\]

Mathematically speaking, this is a problematic definition for several reasons:

1. The partial function \(w\) is defined in terms of itself;
2. It is not clear that such a \(w\) exists; and
3. In case it exists, it is not clear that such a \(w\) is unique.

To see how easily one can yield inappropriate recursive definitions of functions, we refer the reader to the discussion immediately following Theorem 12, which shows examples of recursive definitions which admit no solutions or which are ambiguous.

We next develop the mathematical machinery needed to rigorously define and reason about partial functions like the \(w\) above. More precisely, we frame the mathematics needed here as an instance of the general setting and results discussed in Section 2.6. We strongly encourage the reader to familiarize herself with the definitions and results in Section 2.6 before continuing.

A convenient interpretation of partial functions that may ease the understanding of the subsequent material is as information or knowledge bearers. More precisely, a partial function \(\alpha : \text{State} \rightarrow \text{State}\) can be thought
of as carrying knowledge about some states in $State$, namely exactly those on which $\alpha$ is defined. For such a state $\sigma \in State$, the knowledge that $\alpha$ carries is $\alpha(\sigma)$. If $\alpha$ is not defined in a state $\sigma \in State$ then we can think of it as "$\alpha$ does not have any information about $\sigma$".

Recall from Section 2.6 that the set of partial functions between any two sets can be organized as a bottomed complete partial order (CPO). In our case, if $\alpha, \beta : State \rightarrow State$ then we say that $\alpha$ is less informative than or as informative as $\beta$, written $\alpha \leq \beta$, if and only if for any $\sigma \in State$, it is either the case that $\alpha(\sigma)$ is not defined, or both $\alpha(\sigma)$ and $\beta(\sigma)$ are defined and $\alpha(\sigma) = \beta(\sigma)$. If $\alpha \leq \beta$ then we may also say that $\beta$ refines $\alpha$ or that $\beta$ extends $\alpha$. Then $(State \rightarrow State, \leq \bot)$ is a CPO, where $\bot : State \rightarrow State$ is the partial function which is undefined everywhere.

One can think of each possible iteration of a while loop as an opportunity to refine the knowledge about its denotation. Before the Boolean expression $b$ of the loop while $b$ do $s$ is evaluated the first time, the knowledge that one has about its denotation function $w$ is the empty partial function $\bot : State \rightarrow State$, say $w_0$. Therefore, $w_0$ corresponds to no information.

Now suppose that we evaluate the Boolean expression $b$ in some state $\sigma$ and that it is false. Then the denotation of the while loop should return $\sigma$, which suggests that we can refine our knowledge about $w$ from $w_0$ to the partial function $w_1 : State \rightarrow State$, which is an identity on all those states $\sigma \in State$ for which $[b]\sigma = \text{false}$ and which remains undefined in any other state.

So far we have not considered any state in which the loop needs to evaluate its body. Suppose now that for some state $\sigma$, it is the case that $[b]\sigma = \text{true}$, $[s]\sigma = \sigma'$, and $[b]\sigma' = \text{false}$, that is, that the while loop terminates in one iteration. Then we can extend $w_1$ to a partial function $w_2 : State \rightarrow State$, which, in addition to being an identity on those states on which $w_1$ is defined, that is $w_1 \preceq w_2$, takes each $\sigma$ as above to $w_2(\sigma) = \sigma'$.

By iterating this process, one can define a partial function $w_k : State \rightarrow State$ for any natural number $k$, which is defined on all those states on which the while loop terminates in at most $k$ evaluations of its Boolean condition (i.e., $k - 1$ executions of its body). An immediate property of the partial functions $w_0, w_1, w_2, \ldots, w_k$ is that they increasingly refine each other, that is, $w_0 \preceq w_1 \preceq w_2 \preceq \cdots \preceq w_k$. Informally, the partial functions $w_k$ approximate $w$ as $k$ increases; more precisely, for any $\sigma \in State$, if $w(\sigma) = \sigma'$, that is, if the while loop terminates and $\sigma'$ is the resulting state, then there is some $k$ such that $w_k(\sigma) = \sigma'$. Moreover, $w_n(\sigma) = \sigma'$ for any $n \geq k$.

But the main question still remains unanswered: how to define the denotation $w : State \rightarrow State$ of the looping statement while $(b)$ $s$? According to the intuitions above, $w$ should be some sort of limit of the (infinite) sequence of partial functions $w_0 \preceq w_1 \preceq w_2 \preceq \cdots \preceq w_k \preceq \cdots$. We next formalize all the intuitions above. Let us define the total function

$$\mathcal{F} : (State \rightarrow State) \rightarrow (State \rightarrow State)$$

taking partial functions $\alpha : State \rightarrow State$ to partial functions $\mathcal{F}(\alpha) : State \rightarrow State$ as follows:

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \alpha([s]\sigma) & \text{if } [b]\sigma = \text{true} \\ \sigma & \text{if } [b]\sigma = \text{false} \\ \bot & \text{if } [b]\sigma = \bot \end{cases}$$

The partial functions $w_k$ defined informally above can be now rigorously defined as $\mathcal{F}^k(\bot)$, where $\mathcal{F}^k$ stays for $k$ compositions of $\mathcal{F}$, and $\mathcal{F}^0$ is by convention the identity function, i.e., $1_{(State \rightarrow State)}$ (which is total). Indeed, one can show by induction on $k$ the following property, where $[s]^i$ stays for $i$ compositions of
where the while loop is intended to be undefined. To be more concrete, consider the simple IMP while loop, that is,

\[ \text{while } (k \leq 10) \text{ k = k + 1; } \]

in which the chain stabilizes (which in that case is its LUB and, by Theorem 12, the fixed-point of \( \mathcal{F} \)).

Consider now another fixed-point \( \gamma : \text{State} \rightarrow \text{State} \) of its corresponding \( \mathcal{F} \). While \( \gamma \) must still be the identity on those states \( \sigma \) with \( \sigma(k) \leq 10 \) (indeed, \( \gamma(\sigma) = \mathcal{F}(\gamma)(\sigma) = \sigma \) for any such \( \sigma \in \text{State} \)), it is not enforced to be undefined on any other states. In fact, it can be shown (see Exercise 82) that the fixed-points of \( \mathcal{F} \) are precisely those \( \gamma \) as above with the additional property that \( \gamma(\sigma) = \gamma(\sigma') \) for any \( \sigma, \sigma' \in \text{State} \) with \( \sigma(k) > 10, \sigma'(k) > 10, \) and \( \sigma(x) = \sigma'(x) \) for any \( x \neq k \). Such a \( \gamma \) can be, for example, the following:

\[ \gamma(\sigma) = \begin{cases} \sigma & \text{if } \sigma(k) \leq 10 \\ \iota & \text{otherwise} \end{cases} \]

where \( \iota \in \text{State} \) is some arbitrary but fixed state. Such fixed-points are too informative for our purpose here, since we want the denotation of while to be undefined in all states in which the loop does not terminate. Any other fixed-point of \( \mathcal{F} \) which is strictly more informative than \( \text{fix}(\mathcal{F}) \) is simply too informative.

Second, note that the chain \( \bot \leq \mathcal{F}(\bot) \leq \mathcal{F}^2(\bot) \leq \cdots \leq \mathcal{F}^n(\bot) \leq \cdots \) can be stationary in some cases, but in general it is not. For example, when the loop is well-defined and terminates in any state in some fixed maximum number of iterations which does not depend on the state, its denotation is the (total) function in which the chain stabilizes (which in that case is its LUB and, by Theorem 12, the fixed-point of \( \mathcal{F} \)). For example, the chain corresponding to the loop \( \text{while } (1 <= k && k <= 10) \text{ k = k + 1; } \) stabilizes in 12 steps, each step adding more states to the domain of the corresponding partial function until nothing can be added anymore: at step 1 all states \( \sigma \) with \( \sigma(k) > 10 \) or \( \sigma(k) < 1 \), at step 2 those with \( \sigma(k) = 10 \), at step 3 those with \( \sigma(k) = 9 \), ..., at step 11 those with \( \sigma(k) = 1 \); then no other state is added at step 12, that is, \( \mathcal{F}^{12}(\bot) = \mathcal{F}^{11}(\bot) \). However, the chain associated to a loop is not stationary in general. For example, \( \text{while } (k <= 0) \text{ k = k + 1; } \) terminates in any state, but there is no bound on the number of iterations. Consequently, there is no \( n \) such that \( \mathcal{F}^n(\bot) = \mathcal{F}^{n+1}(\bot) \). Indeed, the later has strictly more information than the former: \( \mathcal{F}^{n+1} \) is defined on all those states \( \sigma \) with \( \sigma(k) = -n \), while \( \mathcal{F}^n \) is not.
3.4.1 The Denotational Semantics of IMP

Figure 3.20 shows the complete denotational semantics of IMP. There is not much to comment on the denotational semantics of the various IMP language constructs, because they have already been discussed above. Note though that the denotation of conjunction captures the desired short-circuited semantics, in that the second conjunct is evaluated only when the first evaluates to true. Also, note that the denotation of programs is still a total function for uniformity (in spite of the fact that some programs may not be well-defined or may not terminate), but one into the CPO State⊥ (see Section 2.6); thus, the denotation of a program which is not well-defined is ⊥. Finally, note that, like in the big-step SOS of IMP in Section 3.2.2, we ignore the non-deterministic evaluation strategies of the + and / arithmetic expression constructs. In fact, since the denotations of the various language constructs are functions, non-deterministic constructs cannot be handled in denotational semantics the same way they were handled in operational semantics. There are ways to deal with non-determinism and concurrency in denotational semantics as discussed at the end of this section, but those are more complex and lead to inefficient interpreters when executed, so we do not consider them in this book. We here limit ourselves to denotational semantics of deterministic languages.

3.4.2 Denotational Semantics in Equational/Rewrite Logic

In order to formalize and execute denotational semantics one needs to formalize and execute the fragment of mathematics that is used by the denotation functions. The size of the needed fragment of mathematics is arbitrary and is typically determined by the particular programming language in question. Since a denotational semantics associates to each program or fragment of program a mathematical object expressed using the formalized language of the corresponding mathematical domain, the faithfulness of any representation/encoding/implementation of denotational semantics into any formalism directly depends upon the faithfulness of the formalizations of the mathematical domains.

The faithfulness of formalizations of mathematical domains is, however, not trivial to characterize in general. The formalization of each mathematical domain requires its own proofs of correctness, and in order for such proofs to make sense we need an alternative, trusted definition of the domain. Consider, for example, the basic domain of natural numbers. One may choose to formalize it using, e.g., Peano-style equational axioms or λ-calculus (see Section 4.5); nevertheless, each of these formalizations needs to be shown correct w.r.t. the mathematical domain of natural numbers, and none of these formalizations is powerful enough to mechanically derive all properties over natural numbers. It is therefore customary when formalizing denotational semantics to simply assume that the formalizations of the mathematical domains are correct; or put differently, to separate the problem of verifying the mathematical domains themselves from the problem of giving a language a denotational semantics using those domains. We here do the same thing: we assume that our equational/rewrite logic formalizations of mathematical domains in Section 2.6.5 which allow us in particular to define and execute higher-order functions and fixed-points for them, are correct. This automatically implies that our equational/rewrite logic representation of denotational semantics is faithful to the original denotational semantics; in other words, unlike for other semantic approaches, no faithfulness theorems for our representation of denotational semantics in equational/rewrite logic are needed.

Denotational Semantics of IMP in Equational/Rewrite Logic

Figure 3.21 shows a direct representation of the denotational semantics of IMP in Figure 3.20 using the mathematical domains formalized in equational/rewrite logic in Section 2.6.5.

To reduce the number of denotation functions defined, we chose to collapse all the syntactic categories under only one sort, Syntax. Similarly, to reuse existing operations of the CPO domain in Section 2.6.5 (e.g.,
Arithmetic expression constructs
\[
[AExp] : AExp \rightarrow (State \rightarrow Int)
\]
\[
[i]_\sigma = i
\]
\[
x]_\sigma = \sigma(x)
\]
\[
[a_1 + a_2]_\sigma = [a_1]_\sigma + [a_2]_\sigma
\]
\[
[a_1 / a_2]_\sigma = \begin{cases} [a_1]_\sigma / [a_2]_\sigma & \text{if } [a_2]_\sigma \neq 0 \\ \bot & \text{if } [a_2]_\sigma = 0 \end{cases}
\]

Boolean expression constructs
\[
[BExp] : BExp \rightarrow (State \rightarrow Bool)
\]
\[
t]_\sigma = t
\]
\[
a_1 \leq a_2]_\sigma = [a_1]_\sigma \leq [a_2]_\sigma
\]
\[
[! b]_\sigma = \neg Bool([b]_\sigma)
\]
\[
[b_1 \& \& b_2]_\sigma = \begin{cases} [b_1]_\sigma & \text{if } [b_1]_\sigma = \text{true} \\ \text{false} & \text{if } [b_1]_\sigma = \text{false} \\ \bot & \text{if } [b_1]_\sigma = \bot \end{cases}
\]

Statement constructs
\[
[Stmt] : Stmt \rightarrow (State \rightarrow State)
\]
\[
[\{\}]_\sigma = \sigma
\]
\[
[\{ s \}]_\sigma = [s]_\sigma
\]
\[
x = a;]_\sigma = \begin{cases} \sigma[[a]_\sigma/x] & \text{if } \sigma(x) \neq \bot \\ \bot & \text{if } \sigma(x) = \bot \end{cases}
\]
\[
[s_1 s_2]_\sigma = [s_2]_\sigma[s_1]_\sigma
\]
\[
[if (b) s_1 \text{ else } s_2]_\sigma = \begin{cases} [s_1]_\sigma & \text{if } [b]_\sigma = \text{true} \\ [s_2]_\sigma & \text{if } [b]_\sigma = \text{false} \\ \bot & \text{if } [b]_\sigma = \bot \end{cases}
\]
\[
[\text{while } (b) s ] = \text{fix}(\mathcal{F}) , \text{ where } \mathcal{F} : (State \rightarrow State) \rightarrow (State \rightarrow State) \text{ defined as}
\]
\[
\mathcal{F}(\alpha)(\sigma) = \begin{cases} \alpha([s]_\sigma) & \text{if } [b]_\sigma = \text{true} \\ \sigma & \text{if } [b]_\sigma = \text{false} \\ \bot & \text{if } [b]_\sigma = \bot \end{cases}
\]

Programs
\[
[Pgm] : Pgm \rightarrow \bot
\]
\[
[int \ x; s] = [s](x \mapsto 0)
\]

Figure 3.20: \textsc{Denot}(IMP): The denotational semantics of IMP.
sort:
Syntax  // generic sort for syntax

subsorts:
AExp, BExp, Stmt, Pgm < Syntax  // syntactic categories fall under Syntax
Int, Bool, State < CPO  // basic domains are regarded as CPOs

operation:
⟦_⟧ : Syntax → CPO  // denotation of syntax

equations:

// Arithmetic expression constructs:
⟦I⟧ = funCPO σ ↦ I
⟦X⟧ = funCPO σ ↦ σ(X)
⟦A1 + A2⟧ = funCPO σ ↦ appCPO(⟦A1⟧, σ) + int appCPO(⟦A2⟧, σ)
⟦A1 / A2⟧ = funCPO σ ↦ ifCPO(appCPO(⟦A2⟧, σ) ≠ int 0, appCPO(⟦A1⟧, σ) / int appCPO(⟦A2⟧, σ), ⊥)

// Boolean expression constructs:
⟦T⟧ = funCPO σ ↦ T
⟦A1 <= A2⟧ = funCPO σ ↦ appCPO(⟦A1⟧, σ) ≤ int appCPO(⟦A2⟧, σ)
⟦! B⟧ = funCPO σ ↦ ¬bool appCPO(⟦B⟧, σ)
⟦B1 & & B2⟧ = funCPO σ ↦ ifCPO(appCPO(⟦B1⟧, σ), appCPO(⟦B2⟧, σ), false)

// Statement constructs:
⟦{}⟧ = funCPO σ ↦ σ
⟦{S}⟧ = funCPO σ ↦ appCPO(⟦S⟧, σ)
⟦X = A;⟧ = funCPO σ ↦ appCPO(funCPO arg ↦ ifCPO(σ(X) ≠ int ⊥, σ[arg/X], ⊥), appCPO(⟦A⟧, σ))
⟦S1 S2⟧ = funCPO σ ↦ appCPO(⟦S2⟧, appCPO(⟦S1⟧, σ))
⟦if (B) S1 else S2⟧ = funCPO σ ↦ ifCPO(appCPO(⟦B⟧, σ), appCPO(⟦S1⟧, σ), appCPO(⟦S2⟧, σ))
⟦while (B) S⟧ = fixCPO funCPO α ↦ funCPO σ ↦ ifCPO(appCPO(⟦B⟧, σ), appCPO(α, appCPO(⟦S⟧, σ)), σ)

// Programs:
⟦int Xl; S⟧ = appCPO(⟦S⟧, (Xl ↦ 0))

Figure 3.21: R_{Desort(IM)}: Denotational semantics of IMP in equational/rewrite logic.
the substitution) on the new mathematical domains without any additional definitional effort, we collapse all
the mathematical domains under CPO; in other words, we now have only one large mathematical domain,
CPO, which includes the domains of integer numbers, Booleans and states as subdomains.

There is not much to say about the equations in Figure 3.21; they restate the mathematical definitions
in Figure 3.20 using our particular CPO language. The equational formalization of the CPO domain in
Section 2.6.5 propagates undefinedness through the CPO operations according to their evaluation strategies.
This allows us to ignore some cases, such as the last case in the denotation of the assignment or the conditional
in Figure 3.20; indeed, if \( \text{app}_{\text{CPO}}(\llbracket A \rrbracket, \sigma) \) is undefined in some state \( \sigma \) then \( \llbracket X = A; \rrbracket \sigma \) will also be undefined,
because \( \text{app}_{\text{CPO}}(\ldots, \bot) \) equals \( \bot \) according to our CPO formalization in Section 2.6.5.

★ Denotational Semantics of IMP in Maude

Figure 3.22 shows the Maude module corresponding to the rewrite theory in Figure 3.21.

3.4.3 Notes

Denotational semantics is the oldest semantic approach to programming languages. The classic paper that
introduced denotational semantics as we know it today, then called “mathematical semantics”, was published
in 1971 by Scott and Strachey [70]. However, Strachey’s interest in the subject started much earlier; he
published two papers in 1966 and 1967, [77] and [78], respectively, which mark the beginning of denotational
semantics. Before Strachey and Scott published their seminal paper [70], Scott published in 1970 a paper
which founded what we call today domain theory [71]. Initially, Scott formalized domains as complete lattices
(which also admit a fixed-point theorem); in time, bottomed complete partial orders (CPOs, see Section 2.6)
turned out to have better properties and they eventually replaced the complete lattices.

We have only used very simple domains in our semantics of IMP in this section, such as domains of
integers, Booleans, and partial functions. Moreover, for simplicity, we are not going to use complex domains
for the IMP++ extension in Section 3.5 either. However, complex languages or better semantics may require
more complex domains. In fact, choosing the right domains is one of the most important aspects of a
denotational semantics. Poor domains may lead to behavioral limitations or to non-modular denotational
semantic definitions. There are two additional important contributions to domain theory that are instrumental
in making denotational semantics more usable:

- **Continuation domains.** The use of continuations in denotational semantics was proposed in 1974, in a
  paper by Strachey and Wadsworth [79]. Wadsworth was the one who coined the term “continuation”, as
  representing “the meaning of the rest of the program”. Continuations allow to have direct access to the
  execution flow, in particular to modify it, as needed for the semantics of abrupt termination, exceptions,
  or call/cc (Scheme was the first language to support call/cc). This way, continuations bring modularity
  and elegance to denotational definitions. However, they come at a price: using continuations affects the
  entire language definition (so one needs to change almost everything) and the resulting semantics are
  harder to read and reason about. There are countless uses of continuations in the literature, not only in
denotational semantics; we refer the interested reader to a survey paper by Reynolds, which details
continuations and their discovery from various perspectives [63].

- **Powerdomains.** The usual domains of partial functions that we used in our denotational semantics of
  IMP are not sufficient to define non-deterministic and/or concurrent languages. Consider, for example,
  the denotation of statements, which are partial functions from states to states. If the language is
  non-deterministic or concurrent, then a statement may take a state into any of many possible different
mod IMP-SEMANTICS-DENOTATIONAL is including IMP-SYNTAX + STATE + CPO .
sort Syntax .
subsorts AExp BExp Stmt Pgm < Syntax .
subsorts Int Bool State < CPO .

op [[_]] : Syntax -> CPO .    --- Syntax interpreted in CPOs

var X : Id . var X1 : List(Id) . var I : Int . var A1 A2 A : AExp .
var T : Bool . var B1 B2 B : BExp . var S1 S2 S : Stmt .
ops sigma alpha arg : -> CPOVar .

eq [[I]] = funCPO sigma -> I .

eq [[X]] = funCPO sigma -> '_('sigma,X') .

eq [[A1 + A2]] = funCPO sigma -> appCPO([[A1]],sigma) +Int appCPO([[A2]],sigma) .

eq [[A1 / A2]] = funCPO sigma -> ifCPO(appCPO([[A2]],sigma) /=Bool 0,
appCPO([[A1]],sigma) /Int appCPO([[A2]],sigma), undefined) .

eq [[T]] = funCPO sigma -> T .

eq [[A1 <= A2]] = funCPO sigma -> appCPO([[A1]],sigma) <=Int appCPO([[A2]],sigma) .

eq [[! B]] = funCPO sigma -> notBool appCPO([[B]],sigma) .

eq [[B1 & B2]] = funCPO sigma -> ifCPO(appCPO([[B1]],sigma),appCPO([[B2]],sigma),false) .

eq [[{}]] = funCPO sigma -> sigma .

eq [[{S}]] = funCPO sigma -> appCPO([[S]],sigma) .

eq [[X = A ;]] = funCPO sigma
-> appCPO(funCPO arg
  -> ifCPO(_('sigma,X) /=Bool undefined, sigma[arg / X], undefined),
  appCPO([[A]],sigma)) .

eq [[S1 S2]] = funCPO sigma -> appCPO([[S2]],appCPO([[S1]],sigma)) .

eq [[if (B) S1 else S2]] = funCPO sigma -> ifCPO(appCPO([[B]],sigma),appCPO([[S1]],sigma),appCPO([[S2]],sigma)) .

eq [[while (B) S]] = fixCPO(funCPO alpha
  -> funCPO sigma
  -> ifCPO(appCPO([[B]],sigma),appCPO(alpha,appCPO([[S]],sigma)),sigma)) .

eq [[(int X1 ; S)]] = appCPO([[S]],(X1 |-> 0)) .
endm

Figure 3.22: The denotational semantics of IMP in Maude
states, or, said differently, it may take a state into a set of states. To give denotational semantics to such languages, Plotkin proposed and formalized the notion of powerdomain \[59\]; the elements of a powerdomain are sets of elements of an underlying domain. Powerdomains make it thus possible to give denotational semantics to non-deterministic languages; used in combination with “resumptions”, powerdomains can also be used to give interleaving semantics to concurrent languages.

As seen above, most of the denotational semantics ideas and principles have been proposed and developed in the 1970s. While it is recommended to read the original papers for historical reasons, some of them may actually use notational conventions and constructions which are not in current use today, making them less accessible. The reader interested in a more modern presentation of denotational semantics and domain theory is referred to Schmidt’s denotational semantics book \[69\], and to Mosses’ denotational semantics chapter \[50\] and Gunter and Scott’s domain theory chapter \[30\] in the Handbook of Theoretical Computer Science (1990).

Denotational semantics are commonly defined and executed using functional languages. A particularly appealing aspect of functional languages is that the domains of partial functions, which are crucial for almost any denotational semantics, and fixed-point operators for them can be very easily defined using the already existing functional infrastructure of these languages; in particular, one needs to define no \( \lambda \)-like calculus as we did in Section 2.6.5. There is a plethora of works on implementing and executing denotational semantics on functional languages. We here only mention Papaspyrou’s denotational semantics of C \[57\], implemented in Haskell; it uses about 40 domains in total and spreads over about 5000 lines of Haskell code. An additional advantage of defining denotational semantics in functional languages is that one can relatively easily port them into theorem provers and then prove properties or meta-properties about them. We refer the interested reader to Nipkow \[55\] for a simple example of how this is done in the context of the Isabelle/HOL prover.

An insightful exercise is to regard domain theory and denotational semantics through the lenses of initial algebra semantics \[26\], which was proposed by Goguen et al. in 1977. The initial algebra semantics approach is simple and faithful to equational logic (Section 2.4); it can be summarized with the following steps:

1. Define a language syntax as an algebraic signature, say \( \Sigma \), which admits an initial (term) algebra \( T_\Sigma \);
2. Define any needed semantic domain as the carrier of corresponding sort in a multi-sorted set, say \( D \);
3. Give \( D \) a \( \Sigma \)-algebra structure, by defining operations corresponding to all symbols in \( \Sigma \);
4. Conclude that the meaning of the language in \( D \) is the unique morphism \( D_t : T_\Sigma \rightarrow D \), which gives meaning \( D_t \) in \( D \) to any fragment of program \( t \).

Let us apply the initial algebra semantics steps above to IMP:

1. \( \Sigma \) is the signature in Figure 3.2.
2. \( D_{AExp} \) is the CPO (\( State \rightarrow Int, \leq, \bot \)) and similarly for the other sorts;
3. \( D_+ : D_{AExp} \times D_{AExp} \rightarrow D_{AExp} \) is the (total) function defined as \( D_+(f_1, f_2)(\sigma) = f_1(\sigma) +_{Int} f_2(\sigma) \) for all \( f_1, f_2 \in D_{AExp} \) and \( \sigma \in State \), and similarly for the other syntactic constructs.
4. The meaning of IMP is given by the unique morphism \( D : T_\Sigma \rightarrow D_{AExp} \).

Therefore, one can regard a denotational semantics of a language as an initial algebra semantics applied to a particular algebra \( D \); in our IMP case, for example, what we defined as \( \ll a \rr \) for \( a \in AExp \) in denotational semantics is nothing but \( D_a \). Moreover, we can now prove our previous denotational semantics definitions; for example, we can prove \( D_{a_1+a_2}(\sigma) = D_{a_1}(\sigma) +_{Int} D_{a_2}(\sigma) \). The above was a very brief account of initial
algebra semantics, but sufficient to appreciate the foundational merits of initial algebra semantics in the context of denotational semantics (initial algebra semantics has many other applications). From an initial algebra semantics perspective, a denotational semantics is all about defining an algebra \( D \) in which the syntax is interpreted. How each fragment gets a meaning follows automatically, from more basic principles.

In that regard, initial algebra semantics achieves a cleaner separation of concerns: syntax is defined as a signature, and semantics is defined as an algebra. There are no equations mixing syntax and semantics, like

\[
\llbracket a_1 + a_2 \rrbracket_\sigma = \llbracket a_1 \rrbracket_\sigma + \llbracket a_2 \rrbracket_\sigma.
\]

We are not aware of any other approaches to define denotational semantics using rewriting and then executing it on rewrite engines as we did in Section 3.4.2. While this is not difficult in principle, as seen in this section, it requires one to give executable rewriting definitions of semantic domains and of fixed points. This may be a tedious and repetitive process on simplistic rewrite engines; for example, the use of membership equational logic, which allows computations to take place also on terms whose intermediate sorts cannot be determined, was crucial for our formalization in Section 2.6.5. Perhaps the closest approach to ours is the one by Goguen and Malcolm in [29]; they define a simple imperative language using the OBJ system (a precursor of Maude) and a style which is a mixture of initial algebra semantics and operational semantics. For example, no fixed-points are used in [29], the loops being simply unrolled like in small-step SOS (see Section 3.3).

### 3.4.4 Exercises

Prove the following exercises, all referring to the IMP denotational semantics in Figure 3.20.

**Exercise 77.** Show the associativity of the addition expression construct, that is, that

\[
\llbracket (a_1 + a_2) + a_3 \rrbracket = \llbracket a_1 + (a_2 + a_3) \rrbracket
\]

for any \( a_1, a_2, a_3 \in AExp \).

**Exercise 78.** Show the associativity of the statement sequential composition, that is, that

\[
\llbracket s_1 (s_2 s_3) \rrbracket = \llbracket (s_1 s_2) s_3 \rrbracket
\]

for any \( s_1, s_2, s_3 \in Stmt \). Compare the elegance of formulating and proving this result using denotational semantics with the similar task using small-step SOS (see Exercise 77).

**Exercise 79.** State and prove the (correct) distributivity property of division over addition.

**Exercise 80.** Prove that the sequential composition statement “\( \text{if} (b) \ s_1 \ \text{else} \ s_2 \ s \)” (i.e., the conditional statement composed sequentially with \( s \)) and the conditional statement “\( \text{if} (b) \{ s \ s \} \ \text{else} \{ s_2 s \} \)” are equivalent, where \( s_1 \) and \( s_2 \) are blocks and where \( s \) is any statement.

**Exercise 81.** Prove that the functions \( F : (State \rightarrow State) \rightarrow (State \rightarrow State) \) associated to IMP while loops satisfy the hypotheses of the fixed-point Theorem 12, so that the denotation of IMP loops is indeed well-defined. Also, prove that the partial functions \( w_k : State \rightarrow State \) defined as

\[
\begin{align*}
\llbracket s \rrbracket_\sigma = \begin{cases} 
\llbracket s \rrbracket_\sigma & \text{if there is } 0 \leq i < k \text{ s.t. } \llbracket b \rrbracket_\sigma = \text{false} \text{ and } \llbracket s \rrbracket_\sigma^{i} \llbracket s \rrbracket_\sigma = \text{true} \text{ for all } 0 \leq j < i \\
\bot & \text{otherwise}
\end{cases}
\end{align*}
\]

are well-defined, that is, that if an \( i \) as above exists then it is unique. Then prove that \( w_k = F^k(\bot) \).

**Exercise 82.** Describe all the fixed-points of the function \( F \) associated to the IMP while loop.
Exercise 83. The semantics in Figure 3.20 evaluates a program performing a division-by-zero to ⊥. Modify the denotational semantics of IMP so that it returns a state like for correct programs, namely the state in which the division-by-zero took place. In other words, we want a variant of IMP where programs fail silently when they perform illegal operations.
should I add definitions for entailments and prove the faithful representation theorem?

add denotational semantics as well in Section 3.5

discuss/add monads as well, as an interesting development besides continuations and powerdomains

use CPO instead of BCPO as an abbreviation for bottomed complete partial orders

make initial algebra semantics into a section

results relating denotational semantics to other semantics
3.5 IMP++: IMP Extended with Several Features

Our goal in this section is to challenge the modularity, reuse capability and flexibility to changes in language design of the basic semantic approaches discussed so far in this chapter, namely big-step SOS, small-step SOS and denotational semantics. We do this by means of a simple programming language design experiment. Specifically, we extend the IMP language in Section 3.1 with several common language features and attempt to give the resulting language a formal semantics following each of the approaches. In each case, we aim at reusing the existing semantics of IMP as much as possible.

Below we describe the features that we add to IMP, together with a short motivation for each:

1. A variable increment operation, \( ++Id \), whose role is to infuse side effects in expressions.

2. An input expression construct, \( \text{read()} \), and an output statement construct, \( \text{print}(AExp); \), whose role is to modify the configurations (one needs to add input/output buffers).

3. Abrupt termination, both by means of an explicit "halt;" statement and by means of implicit division-by-zero, whose role is to enforce a sudden change of the evaluation context.

4. Spawning a new thread with an arithmetic expression construct \( \text{spawn Block} \), which evaluates to a unique thread identifier associated to the newly created thread, and synchronizing with any thread using a "join(AExp);" statement construct. The newly created thread executes the given block concurrently with the rest of the program, sharing all the variables. The thread encountering a join statement gets blocked until the thread it wants to join terminates. The role of thread spawning and joining is to test the support of the various semantic approaches for concurrency.

5. Blocks allowing local variable declarations, \( \{Stmt\} \), where \( Stmt \) can include declarations of the form \( \text{int List\{Id\}}; \) (regarded as ordinary statements). The scope of local declarations is the remainder of the current block. The introduction of blocks with locals requires changing some of the existing syntax and semantics. For example, there is no need for global variable declarations anymore, because they can be replaced by locals. The role of this extension is threefold: (1) to demonstrate how execution environment recovery can be achieved in each semantic approach; (2) to generate some non-trivial feature interactions (e.g., spawned threads share the spawning environment); (3) to highlight a language design scenario where the introduction of a new feature may affect the design of the previous ones.

Two criteria guided us in selecting these particular language extensions: first, these are quite ordinary features encountered in many languages; and second, each of them exposes limitations of one or more of the conventional semantic approaches in this chapter (both before and after this section). We refer to the resulting language extension of IMP as IMP++. Both IMP and IMP++ are admittedly toy languages. However, if a certain programming language semantic approach has difficulties in supporting any of the features of IMP or any of the above IMP extensions in IMP++, then one should most likely expect the same problems, but of course amplified, to occur in practical attempts to define real-life programming languages. By “difficulty” we here mostly mean lack of modularity, that is, in order to define a new feature or to change an existing one we need to make unrelated changes in the already existing semantics of other features. Such formalism-artifact changes are not only tedious and thus inconvenient and demotivate the language designer to extend or experiment with her language, but are also error prone.

IMP++ extends and modifies IMP both syntactically and semantically. Syntactically, it removes from
IMP the global declarations and adds the following constructs:

\[
\begin{align*}
AExp & ::= \quad ++ Id \\
& \quad | \quad \text{read()} \\
& \quad | \quad \text{spawn \ Block} \\
Stmt & ::= \quad \text{print}(AExp); \\
& \quad | \quad \text{halt;} \\
& \quad | \quad \text{join}(AExp); \\
& \quad | \quad \text{int List}\{Id\}; \\
Pgm & ::= \quad Stmt
\end{align*}
\]

Semantically, in addition to defining the new language constructs above, we prefer that division-by-zero implicitly halts the program, same like the explicit use of “\texttt{halt;}”, but in the middle of an expression evaluation. When such an error takes place, one could also generate an error message. However, for simplicity, we do not consider error messages here; we only consider silent termination.

As part of our language design experiment to test the modularity, reuse capability and flexibility to language design changes of the various semantic approaches, we will do the following:

1. We will take each of the IMP++ language features one by one, discussing in detail what it takes to add each of them to IMP making complete abstraction of the other features. In other words, each time we define a feature we pretend that we do not know what other features will be added to the language. Hence, we attempt to achieve local optima for each feature independently. Then, in Section 3.5.6, we put all the features together in the IMP++ language.

2. We will first define \texttt{spawn} as a statement construct, pretending that join thread synchronization were not intended to be an explicit language construct, and will use this variant of \texttt{spawn} when putting together all the features of the IMP++ language in Section 3.5.6. Then, in Section 3.5.7, we add join thread synchronization to the resulting IMP++ language by turning \texttt{spawn} into an expression construct that evaluates to the identifier of the newly created thread, and by adding the “\texttt{join(AExp);}” statement construct that takes the identifier of the thread to join as argument.

We believe that our experiment in this section is not unrealistic. It is quite possible that language designers extend their language one feature at a time, for example to more easily experiment with it or to test it, and that while doing so they would like to only concentrate on the feature at hand and not to think of other possible extensions of the language. It is also reasonable to not worry about thread synchronization from the very beginning when designing a multi-threaded language, particularly because synchronization can also be achieved using shared variables and busy waiting (e.g., using Dekker’s algorithm).

### 3.5.1 Adding Variable Increment

Like in several main-stream programming languages, \texttt{++ \, x} increments the value of \texttt{x} in the state and evaluates to the incremented value. This way, the increment operation makes the evaluation of expressions to now have side effects. Consider, for example, the following two programs:

```
int m, n, s;
int x;
n = 100;
x = 1;
while (++m <= n) { s = s + m; }
x = ++ x / (++ x / x);
```

172
The first program shows that the side effect of variable increment can take place anywhere, even in the condition of a while loop; this is actually quite a common programming pattern in languages with variable increment. The second program shows that the addition of side-effects makes the originally intended evaluation strategies of the various expression constructs important. Indeed, recall that for demonstration purposes we originally wanted \( + \) and \( / \) to be non-deterministic (i.e., to evaluate their arguments stepwise non-deterministically, possibly interleaving their evaluations), while \( \leq \) to be left-to-right sequential. These different evaluation strategies can now lead to different program behaviors. For example, the second program above has five different behaviors! Indeed, the expression assigned to \( x \) can evaluate to 0, 1, 2, 3, and can also perform a division-by-zero. Unfortunately, not all semantic approaches are able to capture all these behaviors.

**Big-Step SOS**

Big-step SOS is one of the semantics which is the most affected by the inclusion of side effects in expressions, because the previous triples \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) and \( \langle b, \sigma \rangle \Downarrow \langle t \rangle \) need to change to four-tuples of the form \( \langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle \) and \( \langle b, \sigma \rangle \Downarrow \langle t, \sigma' \rangle \). These changes are necessary to account for collecting the possible side effects generated by the evaluation of expressions (note that the evaluation of Boolean expressions, because of \( \leq \), can also have side effects). The big-step SOS of almost all the language constructs needs to change as well. For example, the original big-step SOS of division, namely

\[
\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle \quad \text{if } i_2 \neq 0
\]

changes as follows:

\[
\langle a_1, \sigma_1 \rangle \Downarrow \langle i_1, \sigma_1 \rangle \quad \langle a_2, \sigma_1 \rangle \Downarrow \langle i_2, \sigma_2 \rangle \quad \text{if } i_2 \neq 0
\]

\[
\langle a_1, \sigma_2 \rangle \Downarrow \langle i_1, \sigma_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2, \sigma_2 \rangle \quad \text{if } i_2 \neq 0
\]

The rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. We will shortly explain why they fail to fully capture the desired behaviors. One should similarly consider the side effects of expressions in the semantics of statements that need to evaluate expressions. For example, the semantics of the while loop needs to change to propagate the side effects of its condition both when the loop is taken and when the loop is not taken.

Let us next include the big-step semantics of the increment operation; once all the changes to the existing semantics of IMP are applied, the big-step semantics of increment is straightforward:

\[
\langle ++ x, \sigma \rangle \Downarrow \langle \sigma(x) +_\text{mod} 1, \sigma[(\sigma(x) +_\text{mod} 1)/x] \rangle \quad \text{\textbf{BigStep-Inc}}
\]

Indeed, the problem with big-step is not necessarily to define the semantics of variable increment, but what it takes to be able to do it. One needs to redefine configurations as explained above and, consequently, to change the semantics of all the already existing features of IMP to use the new configurations. This, and other features defined later on, show how non-modular big-step semantics is.

In addition to being non-modular, big-step SOS cannot properly deal with non-determinism. While it can capture some limited degree of non-determinism as shown above with \( / \), namely it can non-deterministically choose which of the subexpressions to evaluate first, it cannot define the full non-deterministic strategy (unless we make radical changes to the definition, such as working with sets of values instead of values, which
significantly complicate everything and still fail to capture the non-deterministic behaviors—as it would only capture the non-deterministic evaluation results). To see how the non-deterministic choice evaluation strategy in big-step semantics fails to capture all the desired behaviors, consider the expression \( ++x / (++x / x) \) with \( x \) initially 1, as in the second program at the beginning of Section 3.5.1. This expression can only evaluate to 1, 2 or 3 under non-deterministic choice strategy, like we get in big-step SOS. Nevertheless, as explained at the beginning of Section 3.5.1, it could also evaluate to 0 and even perform a division-by-zero under a fully non-deterministic evaluation strategy, as we will see when we use small-step semantic approaches.

Big-step semantics not only misses behaviors due to its lack of support for non-deterministic evaluation strategies, like shown above, but also hides misbehaviors that it, in principle, detects. For example, assuming \( x > 0 \), the expression \( 1 / (x / ++x) \) can either evaluate to 1 or perform an erroneous division-by-zero. If one searches for all the possible evaluations of a program containing such an expression using the big-step semantics in this section, one will only see the behavior where this expression evaluates to 1; one will never see the erroneous behavior where the division by zero takes place. This will be fixed in Section 3.5.3 where we modify the big-step SOS to support abrupt termination. However, without modifying the semantics, the language designer using big-step semantics may wrongly think that the program is correct. Contrast that with small-step semantics, which, even when one does not add support for abrupt termination, one still detects the wrong behavior by getting stuck on the configuration obtained right before the division by zero.

Additionally, as already explained in Section 3.2.3, the new configurations may be problematic when one wants to execute big-step definitions using rewriting. Indeed, one needs to remove resulting rewrite rules that lead to non-termination, such as rules of the form \( R \rightarrow R \) corresponding to big-step sequents \( R \downarrow R \) where \( R \) are result configurations (e.g., \( \langle i, \sigma \rangle \) with \( i \in \text{Int} \) or \( \langle t, \sigma \rangle \) with \( t \in \{ \text{true}, \text{false} \} \)). We do not use this argument against big-step SOS (its poor modularity is sufficient to disqualify big-step in the competition for an ideal language definitional framework), but rather as a warning to the reader who wants to execute it using rewriting engines (like Maude).

**Type System using Big-Step SOS**

The typing policy of variable increment is the same as that of variable lookup: provided it has been declared, the incremented variable types to an integer. All we need is to add the following typing rule for increment to the already existing typing rules in Figure 3.10.

\[
(xl, x, xl') \vdash ++x : \text{int}
\]  

(BigStepTypeSystem-Inc)

**Small-Step SOS**

Including side effects in expressions is not as bad in small-step semantics as in big-step semantics, because, as discussed in Section 3.3 in small-step SOS one typically uses sequents whose left and right configurations have the same structure even in cases where only some of the configuration components change (e.g., one typically uses sequents of the form \( \langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle \) instead of \( \langle a, \sigma \rangle \rightarrow \langle a' \rangle \); thus, expressions, like any other syntactic categories including statements, can seamlessly modify the state if they need to. However, since we deliberately did not anticipate the inclusion of side effects in expression evaluation, we still have to go back through the existing definition and modify all the rules involving expressions to propagate the side effects. For example, the small-step rule

\[
\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle a_1 / a_2, \sigma \rangle \rightarrow \langle a'_1 / a_2, \sigma \rangle}
\]
for the first argument of / does not apply anymore when the next step in \(a_1\) is an increment operation (since the state \(\sigma\) changes), so it needs to change to

\[
\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma' \rangle
\]

\[
\langle a_1 / a_2, \sigma \rangle \rightarrow \langle a'_1 / a_2, \sigma' \rangle
\]

Of course, all these changes due to side-effect propagation would have not been necessary if we anticipated that side effects may be added to the language, but the entire point of this exercise is to study the strengths of the various semantic approaches without knowing what comes next.

Once all the changes are applied, one can define the small-step SOS of the increment operation almost identically to its big-step SOS (increment is one atomic step, so small equals big):

\[
\langle ++x, \sigma \rangle \rightarrow \langle \sigma(x) + _{Int} 1, \sigma[(\sigma(x) + _{Int} 1)/x] \rangle
\]

(SMALLSTEP-INC)

**Denotational Semantics**

The introduction of expression side effects affects denotational semantics even worse than it affects the big-step SOS. Not only that one has to change the denotation of almost every language construct, but the changes are also heavier and more error prone than for big-step SOS. The first change that needs to be made is the type of the denotation functions for expressions:

\[
\llbracket \_ \rrbracket : AExp \rightarrow (State \rightarrow Int)
\]

\[
\llbracket \_ \rrbracket : BExp \rightarrow (State \rightarrow Bool)
\]

need to change into

\[
\llbracket \_ \rrbracket : AExp \rightarrow (State \rightarrow Int \times State)
\]

\[
\llbracket \_ \rrbracket : BExp \rightarrow (State \rightarrow Bool \times State)
\]

respectively, to take into account the fact that expressions also return a new possibly changed state besides a result when evaluated. Then one has to change the definitions of the denotation functions for each expression construct to propagate the side effects and to properly extract/combine values and states from/in pairs. For example, the previous denotation function of division, where \(\llbracket a_1 / a_2 \rrbracket \sigma\) was defined as

\[
\begin{cases}
\llbracket a_1 \rrbracket \sigma + _{Int} \llbracket a_2 \rrbracket \sigma & \text{if } \llbracket a_2 \rrbracket \sigma \neq 0 \\
\perp & \text{if } \llbracket a_2 \rrbracket \sigma = 0
\end{cases}
\]

needs to change to be defined as

\[
\begin{cases}
(1^{st}(\llbracket a_1 \rrbracket \sigma)) + _{Int} 1^{st}(\llbracket a_2 \rrbracket (2^{nd}(\llbracket a_1 \rrbracket \sigma))), 2^{nd}(\llbracket a_2 \rrbracket (2^{nd}(\llbracket a_1 \rrbracket \sigma)) & \text{if } 1^{st}(\llbracket a_2 \rrbracket (2^{nd}(\llbracket a_1 \rrbracket \sigma))) \neq 0 \\
\perp & \text{if } 1^{st}(\llbracket a_2 \rrbracket (2^{nd}(\llbracket a_1 \rrbracket \sigma))) = 0
\end{cases}
\]

The above is a bit heavy, repetitive and thus error prone. In implementations of denotational semantics, and sometimes even on paper definitions, one typically uses let binders, or \(\lambda\)-abstractions (see Section 4.5), to bind each subexpression appearing more than once in a denotation function to some variable and then using that variable in each place.

In addition to the denotations of expressions, the denotation functions of all statements except for those of the empty block \(\{\}\) and sequential composition also need to change, because they involve expressions and need to take their side effects into account. For example, the denotation of the while loop statement \(\text{while } (b) s\) is the fixed-point of the total function

\[
\mathcal{F} : (State \rightarrow State) \rightarrow (State \rightarrow State)
\]
defined as
\[ F(\alpha)(\sigma) = \begin{cases} 
\alpha(\llbracket s \rrbracket(2^{\text{nd}}(\llbracket b \rrbracket(\sigma)))) & \text{if } 1^{\text{st}}(\llbracket b \rrbracket(\sigma)) = \text{true} \\
2^{\text{nd}}(\llbracket b \rrbracket(\sigma)) & \text{if } 1^{\text{st}}(\llbracket b \rrbracket(\sigma)) = \text{false} \\
\bot & \text{if } \llbracket b \rrbracket(\sigma) = \bot 
\end{cases} \]

All the changes above were necessary to support the side effects generated by the increment construct. The denotational semantics of programs does not need to change. All programs that make no use of increment should still have exactly the same semantics as before (to be precise, the program denotation functions are different as syntactic terms, but they evaluate to the same values when invoked). We are now ready to give the denotational semantics of increment:
\[ \llbracket ++ x \rrbracket(\sigma) = \begin{cases} 
(\sigma(x) + \text{Int}1, \sigma((\sigma(x) + \text{Int}1)/x)) & \text{if } \sigma(x) \neq \bot \\
\bot & \text{if } \sigma(x) = \bot 
\end{cases} \]  
(DENOTATIONAL-Inc)

Like for big-step SOS, giving the denotational semantics of increment is not difficult; the difficulty stays in what it takes to be able to do so.

Needless to say that denotational semantics, as we used it here, is very non-modular. The brute force approach above is the most straightforward approach when one’s goal is to exclusively add increment to IMP—recall that our experiment in this section assumes that each language extension is the last one. When one expects many extensions to a language that in an operational setting would yield changes to the program configuration, in denotational semantics one is better served using a continuation based or a monadic style. These styles were briefly mentioned in Section 3.4.3 and will be further discussed in Section 3.9. They are more involved, and thus less accessible to non-expert language designers. Moreover, switching to such styles is a radical change which requires a complete redefinition of the language. It is therefore highly recommended that one starts directly with a continuation or monadic style if one expects many and non-trivial language extensions. We here, however, prefer the straightforward denotational approach because it is easier to understand and because our overall focus of this book is more operational than denotational.

Besides lacking modularity, the denotational semantics above also lacks non-determinism. Indeed, note that, for example, our denotation of division first evaluates the first expression and then the second expression. Since expressions have side effects, different orders of evaluation can lead to different behaviors. Since the denotations of expressions are partial functions, they cannot have two different behaviors in the same state; therefore, unlike in big-step SOS, we cannot simply add another equation for the other order of evaluation because that would yield an inconsistent mathematical/equational theory, which would violate the basic property of a function to produce no more than one output for any given input. The consecrated method to define non-determinism in denotational semantics is to use powerdomains, as briefly discussed in Section 3.4.3 and further discussed in Section 3.9. Like using continuations or monads, the use of powerdomains also requires a complete redesign of the entire semantics and makes it less accessible to non-experts. Moreover, one cannot obtain a feasible executable model of the language anymore, because the use of powerdomains requires to collect all possible behaviors of any fragment of program at any point in execution; this will significantly slow down the execution of the semantics, making it, for example, infeasible or even unusable as an interpreter anymore.

### 3.5.2 Adding Input/Output

The semantics of the input expression construct `read()` is that it consumes the next integer from the “input buffer” and evaluates to that integer. The semantics of the output statement construct `print(a);` is that `a` is first evaluated to some integer, which is then collected in an “output buffer”. By a “buffer” we here mean a list structure over integers. The semantics of the input/output constructs will be given such that the input
buffer can only be removed integers from its beginning and the output buffer can only be appended integers to its end. If there is no integer left in the input buffer then \textsf{read()} blocks. The output buffer is unbounded, so “\texttt{print(a);}” never blocks when outputting the value of \(a\). Consider the following two programs:

\begin{verbatim}
int m, n, s;
n = \textsf{read}();
while (m <= n)
    { \texttt{print(m); s = s + m}; m = m + 1; }
\texttt{print(s);}
\end{verbatim}

\begin{verbatim}
int s;
s = 0;
while (!\textsf{read}() <= 0)
    { s = s + \textsf{read}(); }
\texttt{print(s);}
\end{verbatim}

The first reads one integer \(i\) from the beginning of the input buffer and then it appends \(i + 2\) integers to the end of the output buffer (the numbers 0, 1, 2, ..., \(i\) followed by their sum). The second reads a potentially unbounded number of integers from the input buffer, terminating if and only if it reads a non-positive integer on an odd position in the input buffer; when that happens, it outputs the sum of the elements on the even positions in the input buffer up to that point.

The addition of \textsf{read()} to IMP means that expression evaluation becomes non-deterministic, regardless of whether we have variable increment or not in our language. Indeed, since / is non-deterministic, an expression of the form \textsf{read()} / \textsf{read()} can evaluate the two reads in any order: for example, if the first two integers in the input buffer are 7 and 3, then this expression can evaluate to either 2 or 0.

Let us formalize buffers. Assume colon-separated integer lists with \(\epsilon\) as identity, \texttt{List}[*\texttt{Int}]*, and let \(\omega, \omega', \omega_1\), etc., range over such lists. The same way we decided for notational convenience to let \texttt{State} be an alias for the map sort \texttt{Map}\((\texttt{Id} \mapsto \texttt{Int})\) (Section 3.1.2), from here on we also let \texttt{Buffer} alias the list sort \texttt{List}[*\texttt{Int}]*.

In a formal language semantics, providing the entire input as part of the initial configuration and collecting the entire output in the result configuration is acceptable, although in implementations of the language one will most likely want the input/output to be interactive. There is some flexibility as to where the input and output buffers should be located in the configuration. One possibility is as new top-level components in the configuration. Another possibility is as special variables in the already existing state. The latter would require some non-trivial changes in the mathematical model of the state, so we prefer to follow the former approach in the sequel. An additional argument in favor of our choice is that sooner or later one needs to add new components to the configuration anyway, so we take this opportunity to discuss how robust/modular the various semantic styles are with regards to changes in the structure of the configuration.

**Big-Step SOS**

To accommodate the input/output buffers, all configurations and all sequents we had in the original big-step SOS of IMP in Sections 3.2.1 and 3.2.2 need to change. For example, since expressions can now consume input, the original expression sequents of form \(\langle a, \sigma \rangle \Downarrow \langle i \rangle\) and \(\langle b, \sigma \rangle \Downarrow \langle t \rangle\) need to change into sequents \(\langle a, \sigma, \omega \rangle \Downarrow \langle i, \omega' \rangle\) and \(\langle b, \sigma, \omega \rangle \Downarrow \langle t, \omega' \rangle\) (recall that we add one feature at a time, so expression evaluation currently does not have side effects on the state), respectively, where \(\omega, \omega' \in \texttt{Buffer}\). Also, the big-step SOS rules for expressions need to change to take into account both the new configurations and the fact that expression evaluation can now affect the input buffer. For example, the original big-step SOS of division,

\[\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle i_1 + \text{mod} i_2 \rangle} \text{ if } i_2 \neq 0\]
changes as follows:

\[
\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle i_1, \omega_1 \rangle \quad \langle a_2, \sigma, \omega_1 \rangle \Downarrow \langle i_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle i_1 / i_2, \omega_1, \omega_2 \rangle} \quad \text{if } i_2 \neq 0
\]

\[
\frac{\langle a_1, \sigma, \omega_1 \rangle \Downarrow \langle i_1, \omega_1 \rangle \quad \langle a_2, \sigma, \omega_1 \rangle \Downarrow \langle i_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle i_1 / i_2, \omega_1 \rangle} \quad \text{if } i_2 \neq 0
\]

Like for the variable increment, the rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. Unfortunately, they also only capture a non-deterministic choice strategy, failing to capture the intended full non-determinism of division.

Since statements can both consume input and produce output, their big-step SOS sequents need to change from \(\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle\) to \(\langle s, \sigma, \omega_{in} \rangle \Downarrow \langle \sigma', \omega_{in}', \omega_{out} \rangle\), where \(\omega_{in}, \omega_{in}' \in Buffer\) are the input buffers before and, respectively, after the evaluation of statement \(s\), and where \(\omega_{out} \in Buffer\) is the output produced during the evaluation of \(s\). Unfortunately, all big-step SOS rules for statements also have to change, to accommodate the additional input and/or output components in configurations. For example, the semantics of sequential composition needs to change from

\[
\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle \quad \langle s_2, \sigma_1 \rangle \Downarrow \langle \sigma_2 \rangle}{\langle s_1 \ s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle}
\]

to

\[
\frac{\langle s_1, \sigma, \omega_{in} \rangle \Downarrow \langle \sigma_1, \omega_{in}', \omega_{out}' \rangle \quad \langle s_2, \sigma_1, \omega_{in}' \rangle \Downarrow \langle \sigma_2, \omega_{in}', \omega_{out} \rangle}{\langle s_1 \ s_2, \sigma, \omega_{in} \rangle \Downarrow \langle \sigma_2, \omega_{in}', \omega_{out} \rangle}
\]

Note that the outputs of \(s_1\) and of \(s_2\) have been appended in order to yield the output of \(s_1 \ s_2\).

Finally, we also have to change the initial configuration holding programs to also take an input, as well as the big-step SOS rule for programs from

\[
\frac{\langle s, x l \mapsto 0 \rangle \Downarrow \langle \sigma \rangle \quad \langle \text{int } x l; \ s \rangle \Downarrow \langle \sigma \rangle}{\langle \text{int } x l; \ s \rangle \Downarrow \langle \sigma \rangle}
\]

into

\[
\frac{\langle s, x l \mapsto 0, \omega_{in} \rangle \Downarrow \langle \sigma, \omega_{in}', \omega_{out} \rangle \quad \langle \text{int } x l; \ s, \omega_{in} \rangle \Downarrow \langle \sigma, \omega_{in}', \omega_{out} \rangle}{\langle \text{int } x l; \ s, \omega_{in} \rangle \Downarrow \langle \sigma, \omega_{in}', \omega_{out} \rangle}
\]

We keep the input buffer in the final configuration for two reasons: to avoid having to define a new configuration holding only the state and the output, and to allow the language designer to more easily debug her semantics, in particular to see whether there is any input left unused at the end of the program. One can argue that now, since we have output in our language, the result configuration may actually contain only the output (that is, it can also drop the state). The reader is encouraged to experiment with different final configurations.

Hence, almost everything changed in the original big-step SOS of IMP in order to prepare for the addition of input/output. All theses necessary changes highlight, again, the lack of modularity of big-step SOS. Once all the changes above are applied, one can easily define the semantics of the input/output constructs:

\[
\langle \text{read}(), \sigma, i : \omega_{in} \rangle \Downarrow \langle i, \omega_{in} \rangle \quad \text{(BigStep-Read)}
\]

\[
\frac{\langle a, \sigma, \omega_{in} \rangle \Downarrow \langle i, \omega'_{in} \rangle}{\langle \text{print}(a);, \sigma, \omega_{in} \rangle \Downarrow \langle \sigma, \omega'_{in}, i \rangle} \quad \text{(BigStep-Print)}
\]
Type System using Big-Step SOS

The typing policy of the input/output constructs is straightforward, though recall that we decided to only allow to read and to print integers. To type programs using input/output, we therefore add the following typing rules to the already existing typing rules in Figure 3.10:

\[
\begin{align*}
\text{xl} & \vdash \text{read()} : \text{int} & \text{(BigStepTypeSystem-Read)} \\
\text{xl} & \vdash a : \text{int} \quad \text{xl} \vdash \text{print}(a) ; : \text{stmt} & \text{(BigStepTypeSystem-Print)}
\end{align*}
\]

Small-Step SOS

Like in big-step SOS, in small-step SOS we also need to change all IMP’s configurations in Section 3.3.1 in order to add input/output. In the spirit of making only minimal changes, we modify the configurations holding expressions to also hold an input buffer, and the configurations holding statements to also hold both an input and an output buffer. Implicitly, all IMP’s small-step SOS rules in Section 3.3.2 also need to change. The changes are straightforward, essentially having to just propagate the output through each statement construct, but they are still changes and thus expose, again, the lack of modularity of small-step SOS. Here is, for example, how the small-step SOS rules for division and for sequential composition need to change:

\[
\begin{align*}
\langle a_1, σ, ω_{\text{in}} \rangle & \rightarrow \langle a'_1, σ, ω'_{\text{in}} \rangle \\
\langle a_1 / a_2, σ, ω_{\text{in}} \rangle & \rightarrow \langle a'_1 / a'_2, σ, ω'_{\text{in}} \rangle \\
\langle a_2, σ, ω_{\text{in}} \rangle & \rightarrow \langle a'_2, σ, ω'_{\text{in}} \rangle \\
\langle a_1 / a_2, σ, ω_{\text{in}} \rangle & \rightarrow \langle a'_1 / a'_2, σ, ω'_{\text{in}} \rangle \\
\langle s_1, σ, ω_{\text{in}}, ω_{\text{out}} \rangle & \rightarrow \langle s'_1, σ', ω'_{\text{in}}, ω'_{\text{out}} \rangle \\
\langle s_1 s_2, σ, ω_{\text{in}}, ω_{\text{out}} \rangle & \rightarrow \langle s'_1 s'_2, σ', ω'_{\text{in}}, ω'_{\text{out}} \rangle
\end{align*}
\]

The expression configurations do not need to consider an output buffer because, as already discussed, in this language design experiment we assume at each stage only the current feature, without attempting to anticipate other features that will be possibly added in the future, and we attempts to do minimal changes. For example, if functions were to be added to the language later, in which case expressions will also possibly affect the output through function calls, then all the expression configurations and their corresponding small-step SOS rules will need to change again.

Finally, we also have to change the initial configuration holding programs to also take an input, as well as the small-step SOS rule for programs to initialize the output buffer to $\epsilon$ as follows:

\[
\langle \text{int xl}; s, ω_{\text{in}} \rangle \rightarrow \langle s, (\text{xl} \mapsto 0), ω_{\text{in}}, \epsilon \rangle
\]

Once all the changes are applied, we can give the small-step SOS of input/output as follows:

\[
\begin{align*}
\langle \text{read()}, σ, i : ω_{\text{in}} \rangle & \rightarrow \langle i, σ, ω_{\text{in}} \rangle & \text{(SmallStep-Read)} \\
\langle a, σ, ω_{\text{in}} \rangle & \rightarrow \langle a', σ, ω'_{\text{in}} \rangle & \text{(SmallStep-Print-Arg)} \\
\langle \text{print}(a) ; ; σ, ω_{\text{in}}, ω_{\text{out}} \rangle & \rightarrow \langle \text{print}(a') ; ; σ, ω'_{\text{in}}, ω_{\text{out}} \rangle \\
\langle \text{print}(i) ; ; σ, ω_{\text{in}}, ω_{\text{out}} \rangle & \rightarrow \langle \{ i \}, σ, ω_{\text{in}}, ω_{\text{out}} : i \rangle & \text{(SmallStep-Print)}
\end{align*}
\]
Denotational Semantics

To accommodate the input and the output buffers, the denotation functions associated to IMP’s syntactic categories need to change their types from

\[
\begin{align*}
\llbracket \cdot \rrbracket &: AExp \rightarrow (State \rightarrow Int) \\
\llbracket \cdot \rrbracket &: BExp \rightarrow (State \rightarrow Bool) \\
\llbracket \cdot \rrbracket &: Stmt \rightarrow (State \rightarrow State) \\
\llbracket \cdot \rrbracket &: Pgm \rightarrow State_{\bot}
\end{align*}
\]

to

\[
\begin{align*}
\llbracket \cdot \rrbracket &: AExp \rightarrow (State \times Buffer \rightarrow Int \times Buffer) \\
\llbracket \cdot \rrbracket &: BExp \rightarrow (State \times Buffer \rightarrow Bool \times Buffer) \\
\llbracket \cdot \rrbracket &: Stmt \rightarrow (State \times Buffer \rightarrow State \times Buffer \times Buffer) \\
\llbracket \cdot \rrbracket &: Pgm \rightarrow (Buffer \rightarrow State \times Buffer \times Buffer)
\end{align*}
\]

We next discuss the definitions of the new denotation functions.

The denotations of expressions now take a state and an input buffer and produce a value and a possibly modified input buffer. For example, the denotation of division becomes:

\[
\llbracket a_1 / a_2 \rrbracket_\pi = \begin{cases} 
(1^\text{st}(\text{arg}_1) \mathbin{/} \text{int} 1^\text{st}(\text{arg}_2), \ 2^\text{nd}(\text{arg}_2)) & \text{if } 1^\text{st}(\text{arg}_2) \neq 0 \\
 \bot & \text{if } 1^\text{st}(\text{arg}_2) = 0
\end{cases}
\]

where \(\text{arg}_1 = \llbracket a_1 \rrbracket_\pi\) and \(\text{arg}_2 = \llbracket a_2 \rrbracket(1^\text{st}(\pi), 2^\text{nd}(\text{arg}_1))\).

The denotations of statements can now produce an output buffer in addition to a modified input buffer (and a state). For example, the denotation of sequential composition becomes:

\[
\llbracket s_1 \ s_2 \rrbracket_\pi = (1^\text{st}(\text{arg}_2), \ 2^\text{nd}(\text{arg}_2), \ 3^\text{rd}(\text{arg}_1) : 3^\text{rd}(\text{arg}_2))
\]

where \(\text{arg}_1 = \llbracket s_1 \rrbracket_\pi\) and \(\text{arg}_2 = \llbracket s_2 \rrbracket(1^\text{st}(\text{arg}_1), 2^\text{nd}(\text{arg}_1))\). As another example of statement denotational semantics, the denotational semantics of while loops remains a fixed-point, but in order to be consistent with the new type of the denotation function for statements, it needs to be the fixed point of a (total) function

\[
\mathcal{F} : (State \times Buffer \rightarrow State \times Buffer \times Buffer) \rightarrow (State \times Buffer \rightarrow State \times Buffer \times Buffer)
\]

It is not difficult to see that the following definition of \(\mathcal{F}\) has the right type and that \(\mathcal{F}(\alpha)\) indeed captures the information that is added to \(\alpha\) by unrolling the loop \texttt{while} \(b\) \texttt{s} once:

\[
\mathcal{F}(\alpha)(\pi) = \begin{cases} 
(1^\text{st}(\text{arg}_3), \ 2^\text{nd}(\text{arg}_3), \ 3^\text{rd}(\text{arg}_2) : 3^\text{rd}(\text{arg}_3)) & \text{if } 1^\text{st}(\text{arg}_1) = \text{true} \\
(1^\text{st}(\pi), \ 2^\text{nd}(\text{arg}_1), \ e) & \text{if } 1^\text{st}(\text{arg}_1) = \text{false} \\
\bot & \text{if } \text{arg}_1 = \bot
\end{cases}
\]

where \(\text{arg}_1 = \llbracket b \rrbracket_\pi\), \(\text{arg}_2 = \llbracket s \rrbracket(1^\text{st}(\pi), 2^\text{nd}(\text{arg}_1))\), and \(\text{arg}_3 = \alpha(1^\text{st}(\text{arg}_2), \ 2^\text{nd}(\text{arg}_2))\).

Programs now take an input as well; like for big-step SOS, we prefer to also make the remaining input available at the end of the program execution, in addition to the state and the output:

\[
\llbracket \text{int } xl; \ s \rrbracket_\omega = \llbracket s \rrbracket((xl \mapsto 0), \omega)
\]

Once all the changes on the denotations of the various syntactic categories are applied as discussed above, adding the semantics of the new input/output constructs is immediate:
\[ \text{[read()]\pi} = (\text{head}(2^{nd}(\pi)), \text{tail}(2^{nd}(\pi))) \]  
(DENOTATIONAL-READ)

\[ \text{[print(a);\pi} = (1^{st}(\pi), 2^{nd}([a]\pi), 1^{st}([a]\pi)) \]  
(DENOTATIONAL-PRINT)

Like in the case of IMP’s extension with the variable increment expression construct, the denotational semantics of the extension with the input/output constructs would have been more modular if we had adopted a continuation or monadic style from the very beginning.

3.5.3 Adding Abrupt Termination

IMP++ adds both implicit and explicit abrupt program termination. The implicit one is given by division by zero, while the explicit abrupt termination is given by a new statement added to the language, “halt;”. For uniformity and demonstration purposes, we choose a silent termination semantics. That is, in both cases of abrupt termination, the resulting configuration has the same structure as if the program terminated normally; for example, in the case of big-step SOS, we would like the result configuration for statements to be \langle \sigma \rangle, where \sigma is the state when the program was terminated abruptly. Specifically, we want the programs

```plaintext
int m, n, s;
    n = 100;
while (true) {
    if (m <= n) {
        s = s + m;
        m = m + 1;
    } else { halt; }
}
```

```plaintext
int m, n, s;
    n = 100;
while (true) {
    if (m <= n) {
        s = s + m;
        m = m + 1;
    } else { s = s / (n / m); }
}
```

to yield the result configuration \langle m |\rightarrow 101 & n |\rightarrow 100 & s |\rightarrow 5050 \rangle instead of a special configuration of the form \langle halting, m |\rightarrow 101 & n |\rightarrow 100 & s |\rightarrow 5050 \rangle or similar. Unfortunately, that is not possible in all cases without intrusively modifying the syntax of the IMP language (to catch the exceptional behavior and explicitly discard the additional information), since some operational styles need to make a sharp distinction between a halting configuration and a normal configuration (for propagation reasons). Proponents of those semantic styles may argue that our semantic choice above seems inappropriate, since giving more information in the result configuration, such as “this is a halting configuration”, is better for all purposes than giving less information. There are, however, also reasons to always want a normal result configuration upon termination. For example, one may want to include IMP in a larger context, such as in a distributed system, where all the context wants to know about the embedded language is that it takes a statement and produces a state and/or an output; IMP’s internal exceptional situations are of no concern to the outer context. There is no absolute better or worse language design, both in what regards syntax and in what regards semantics. Our task here is to make the language designer aware of the subtleties and the limitations of the various semantic approaches, and not to propose a better programming language.

Big-Step SOS

The lack of modularity of big-step semantics will be, again, emphasized here. Let us first add the semantic definition for the implicit abrupt termination generated by division by zero. Recall that the big-step SOS rule for division was the following:

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle i_1 +_{tw} i_2 \rangle} \text{ if } i_2 \neq 0
\]
We keep that unchanged, but we also add the following new rule:

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle 0 \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle \text{error} \rangle}
\]

\text{(BigStep-Div-By-Zero)}

In the above rule, \text{error} can be regarded as a special value; alternatively, one can regard \langle \text{error} \rangle as a special result configuration.

But what if the evaluation of \( a_1 \) or of \( a_2 \) in the above rule generates itself an error? If that is the case, then one needs to propagate that error through the division construct:

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle \text{error} \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle \text{error} \rangle}
\]

\text{(BigStep-Div-Error-Left)}

\[
\frac{\langle a_2, \sigma \rangle \Downarrow \langle \text{error} \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle \text{error} \rangle}
\]

\text{(BigStep-Div-Error-Right)}

Note that in case one of \( a_1 \) or \( a_2 \) generates an error, then the other one is not even evaluated anymore, to faithfully capture the intended meaning of abrupt termination.

Unfortunately, one has to do the same for all the expression language constructs. This way, for each expression construct, one has to add at least as many error-propagation big-step SOS rules as arguments that expression construct takes. Moreover, when the evaluation error reaches a statement, one needs to transform it into a “halting signal”. This can be achieved by introducing a new type of result configuration, namely \langle \text{halting}, \sigma \rangle, and then adding appropriate halting propagation rules for all the statements. For example, the assignment statement needs to be added the new rule

\[
\frac{\langle a, \sigma \rangle \Downarrow \langle \text{error} \rangle}{\langle x = a; \sigma \rangle \Downarrow \langle \text{halting}, \sigma \rangle}
\]

\text{(BigStep-Asgn-Halt)}

The halting signal needs to be propagated through statement constructs, collecting the appropriate state. For example, the following two rules need to be included for sequential composition, in addition to the existing rule (which stays unchanged):

\[
\frac{\langle s_1, \sigma \rangle \Downarrow \langle \text{halting}, \sigma_1 \rangle}{\langle s_1 s_2, \sigma \rangle \Downarrow \langle \text{halting}, \sigma_1 \rangle}
\]

\text{(BigStep-Seq-Halt-Left)}

\[
\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle, \langle s_2, \sigma_1 \rangle \Downarrow \langle \text{halting}, \sigma_2 \rangle}{\langle s_1 s_2, \sigma \rangle \Downarrow \langle \text{halting}, \sigma_2 \rangle}
\]

\text{(BigStep-Seq-Halt-Right)}

In addition to all the propagation rules, we also have to define the semantics of the explicit halt statement:

\[
\langle \text{halt};, \sigma \rangle \Downarrow \langle \text{halting}, \sigma \rangle
\]

\text{(BigStep-Halt)}

Therefore, when using big-step SOS, one has to more than \textit{double} the number of rules in order to support abrupt termination. Indeed, any argument of any language construct can yield the termination signal, so a rule is necessary to propagate that signal through the current language construct. It is hard to imagine worse in a language design framework. An unfortunate language designer choosing big-step semantics as language definition framework will incrementally become reluctant to add or experiment with any new feature in her language. For example, imagine that one wants to add exceptions and break/continue of loops to IMP++.
Finally, unless one extends the language syntax, there appears to be no way to get rid of the junk result configurations \((\text{halting}, \sigma')\) that have been artificially added in order to propagate the error or the halting signals. For example, one cannot simply add the rule

\[
\langle s, \sigma \rangle \Downarrow (\text{halting}, \sigma') \\
\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle
\]

because it may interfere with other rules and thus wrongly hide the halting signal; for example, it can be applied on the second hypothesis of the rule \((\text{BigStep-Seq-Halt-Right})\) above hiding the halting signal and thus wrongly making the normal rule \((\text{BigStep-Seq})\) applicable. While having junk result configurations of the form \((\text{halting}, \sigma)\) may seem acceptable in our scenario here, perhaps even desirable for debugging reasons, in general one may find it inconvenient to have many types of result configurations; indeed, one would need similar junk configurations for exceptions, for break/continue of loops, for functions return, etc.

Consequently, the halting signal needs to be caught at the top-level of the derivation. Fortunately, IMP provides a top-level syntactic category, \(Pgm\), so we can add the following rule which dissolves the potential junk configuration at the top:

\[
\langle s, xl \mapsto 0 \rangle \Downarrow (\text{halting}, \sigma) \\
\langle \text{int} \; xl; \; s \rangle \Downarrow \langle \sigma \rangle \quad (\text{BigStep-Halt})
\]

Now we have silent abrupt termination, but note that the variable declaration construct now acts as an exception catching and dissolving the abrupt termination signal generated by “halt;” or by division-by-zero. If one does not like to use a language construct for something which has not been originally intended, then one can add an auxiliary \(\text{top}\) statement or program construct, then reduce the semantics of programs to that of \(\text{top}\), and then give \(\text{top}\) an exception-handling-like big-step SOS, as we will do for the MSOS definition of abrupt termination in Section \[5.6\]; see Exercise \[93\]. This latter solution is also more general, because it does not rely on a fortunate earlier decision to have a top-level language construct.

In addition to the lack of modularity due to having to more than double the number of rules in order to add abrupt termination, the inclusion of all these rules can also have a significant impact on performance when one wants to execute the big-step SOS. Indeed, there are now four rules for division, each having the same left-hand-side, \(\langle a_1 / a_2, \sigma \rangle\), and some of these rules even sharing some of the hypotheses. That means that any general-purpose proof or rewrite system attempting to execute such a definition will unavoidably face the problem of searching a large space of possibilities in order to find one or all possible reductions.

**Type System using Big-Step SOS**

The typing policy of abrupt termination is clear: “halt;” types to a statement and division-by-zero is ignored. Indeed, one cannot expect that a type checker, or any technique, procedure or algorithm, can detect division by zero in general: division-by-zero, like almost any other runtime property of any Turing-complete programing language, is an undecidable problem. Consequently, it is common to limit typing division to checking that the two expressions have the expected type, integer in our case, which our existing type checker for IMP already does (see Figure \[3.10\]). We therefore only add the following typing rule for “halt;”:

\[
xl \vdash \text{halt; : stmt} \quad (\text{BigStepTypeSystem-Halt})
\]

**Small-Step SOS**

Small-step SOS turns out to be almost as non-modular as big-step SOS when defining control-intensive constructs like abrupt termination. Like for big-step SOS, we need to invent special configurations to signal
steps corresponding to implicit division by zero or to explicit halt statements. In small-step SOS, a single type of such special configurations suffices, namely one of the form \((\text{halting}, \sigma)\), where \(\sigma\) is the state in which the program was abruptly terminated. However, for uniformity with big-step SOS, we also define two types of special configurations, one for expressions and one for statements; since we carry the state in the right-hand-side configuration of all sequents in our small-step SOS definitions, the only difference between the two configuration types is their tag, namely \((\text{error}, \sigma)\) for the former versus \((\text{halting}, \sigma)\) for the latter.

We can then define the small-step SOS of division by zero as follows (recall that the original SmallStep-Div rule in Figure 3.14 is \(\langle i_1 \div i_2, \sigma \rangle \rightarrow \langle i_1 \divo i_2, \sigma \rangle \) if \(i_2 \neq 0\)):

\[
\langle i_1 \div 0, \sigma \rangle \rightarrow \langle \text{error}, \sigma \rangle \\
\langle a_1 \div a_2, \sigma \rangle \rightarrow \langle \text{error}, \sigma \rangle
\]

Like for the big-step SOS extension above, we have to make sure that the halting signal is correctly propagated. Here are, for example, the propagation rules through the division construct:

\[
\langle a_1, \sigma \rangle \rightarrow \langle \text{error}, \sigma \rangle \\
\langle x = a_1; , \sigma \rangle \rightarrow \langle \text{halting}, \sigma \rangle
\]

The two rules above are given in such a way that the semantics is faithful to the intended computational granularity of the defined language feature. Indeed, we want division by zero to take one computational step to be reported as an error, as opposed to as many steps as the depth of the context in which the error has been detected; for example, a configuration containing expression \((3 \div 0) \div 3\) should reduce to a halting configurations in one step, not in two. If we added a special error value and replaced the two rules above by

\[
\langle \text{error} / a_2, \sigma \rangle \rightarrow \langle \text{error}, \sigma \rangle \\
\langle a_1 / \text{error}, \sigma \rangle \rightarrow \langle \text{error}, \sigma \rangle
\]

then errors would be propagated to the top level of the program in as many small-steps as the depth of the context in which the error was generated; we do not want that.

Like in the big-step SOS above, the implicit expression errors need to propagate through the statements and halt the program. One way to do it is to generate an explicit \texttt{halt;} statement and then to propagate that \texttt{halt;} statement through all the statement constructs as if it was a special statement value, until it reaches the top. However, as discussed in the paragraph above, that would generate as many steps as the depth of the evaluation contexts in which the \texttt{halt;} statement is located, instead of just one step as desired. Alternatively, we can use the same approach to propagate the halting configuration through the statement constructs as we used to propagate it through the expression constructs. Specifically, we add transition rules from expressions to statements, like the one below (a similar one needs to be added for the conditional):

\[
\langle a, \sigma \rangle \rightarrow \langle \text{error}, \sigma \rangle \\
\langle x = a; , \sigma \rangle \rightarrow \langle \text{halting}, \sigma \rangle
\]

Once the halting signal due to a division by zero reaches the statement level, it needs to be further propagated through the sequential composition, that is, we need to add the following rule:

\[
\langle s_1, \sigma \rangle \rightarrow \langle \text{halting}, \sigma \rangle \\
\langle s_1 ; s_2, \sigma \rangle \rightarrow \langle \text{halting}, \sigma \rangle
\]
Note that we assumed that a halting step does not change the state (we used the same $\sigma$ in both the left and the right configurations). One can prove by structural induction that in our simple language scenario that is indeed the case, so there is no need to propagate state changes when the program halts.

Finally, we can also generate a special halting configuration when a “halt;” statement is reached:

$$\langle \text{halt};, \sigma \rangle \rightarrow \langle \text{halting}, \sigma \rangle$$

(SmallStep-Halt)

Now any abruptly terminated program reduces to a special configuration of the form $\langle \text{halting}, \sigma \rangle$. Recall that our plan, however, was to terminate the computation with a normal configuration of the form $\langle \{\}, \sigma \rangle$, regardless of whether the program terminates normally or abruptly. Like in the big-step SOS above, the naive solution to transform a step producing a halting configuration into a normal step using the rule

$$\langle s, \sigma \rangle \rightarrow \langle \text{halting}, \sigma \rangle$$

$$\langle s, \sigma \rangle \rightarrow \langle \{\}, \sigma \rangle$$

does not work. Indeed, consider a situation where the rule (SmallStep-Seq-Halt) above could apply. There is nothing to prevent the naive rule above to interfere and transform the halting premise of (SmallStep-Seq-Halt) into a normal step producing a $\{\}$, which can be further fed to the conventional rule for sequential composition, (SmallStep-Seq-Arg1) in Figure 3.15 hereby continuing the execution of the program as if no abrupt termination took place.

If one is willing to waste a computational step in order to explicitly dissolve the halting configuration replacing it by a normal one, then one can add the following simple small-step SOS rule:

$$\langle \text{halting}, \sigma \rangle \rightarrow \langle \{\}, \sigma \rangle$$

(SmallStep-Halting)

Although it may look like this rule has the same final effect as the wrong conditional conditional rule above, in the sense that it reduces the halting statement in two steps instead of one to the empty-block configuration, note that this difference in the number of steps is actually crucial here! The above works because small-step SOS performs only one step at a time, both globally and in the rule premises.

Although not perfect, because it wastes one step, we find that the rule above gives us a good trade-off between elegance and computational intrusiveness (after all, wasting a step in a deterministic manner may be acceptable in many situations). Supposing that one wants to waste no computational steps as an artifact of the particular small-step SOS approach chosen, there is no immediate way to terminate the program with a normal result configuration of the form $\langle \{\}, \sigma \rangle$ both when the program terminates abruptly and when it terminates normally. One possibility, also suggested for big-step SOS above and followed in the subsequent MSOS definition for abrupt termination in Section 3.6, is to add an auxiliary top language construct. With that, we can change the small-step SOS rule for variable declarations to reduce the top-level program to its body statement wrapped under this top construct, and then add corresponding rules to propagate normal steps under the top while catching the halting steps and transforming them into normal steps at the top-level.

Here are four small-step SOS rules which almost achieve this (the first rule replaces the previous one for variable declarations); see also Exercise 96:

$$\langle \text{int} x l; s \rangle \rightarrow \langle \text{top} s, (x l \mapsto 0) \rangle$$

$$\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$$

$$\langle \text{top} s, \sigma \rangle \rightarrow \langle \text{top} s', \sigma' \rangle$$

$$\langle \text{top} \{\}, \sigma \rangle \rightarrow \langle \{\}, \sigma \rangle$$

$$\langle s, \sigma \rangle \rightarrow \langle \text{halting}, \sigma \rangle$$

$$\langle \text{top} s, \sigma \rangle \rightarrow \langle \{\}, \sigma \rangle$$

185
We said the rules above “almost” achieve non-wasteful reduction because the third rule still wastes one step to eliminate the unnecessary top construct once its statement has been reduced. This rule is easy to avoid though. All we have to do is to add an additional case for each of the first two rules, namely a case where the argument statement of the top construct is {}, and in that case to replace top {} with {}; see also Exercise 97.

Denotational Semantics

Since both expressions and statements can abruptly terminate, like in the previous semantics we have to provide a means for the denotation functions for expressions and statements to flag when abrupt termination is intended. This way, the denotation of programs can catch the abrupt termination flag and yield the expected state (recall that we want to see normal termination of programs regardless of whether that happens abruptly or not). Specifically, we change the previous denotation functions

\[
\llbracket \cdot \rrbracket : AExp \rightarrow (State \rightarrow Int) \\
\llbracket \cdot \rrbracket : BExp \rightarrow (State \rightarrow Bool) \\
\llbracket \cdot \rrbracket : Stmt \rightarrow (State \rightarrow State)
\]

into denotation functions of the form

\[
\llbracket \cdot \rrbracket : AExp \rightarrow (State \rightarrow Int \cup \{error\}) \\
\llbracket \cdot \rrbracket : BExp \rightarrow (State \rightarrow Bool \cup \{error\}) \\
\llbracket \cdot \rrbracket : Stmt \rightarrow (State \rightarrow State \times \{halting, ok\})
\]

as described below. Before we proceed, note that the new denotation functions will still associate partial functions to syntactic categories. While the new semantics will be indeed able now to catch and terminate when a division by zero takes place, it will still not be able to catch non-termination of programs; the denotation of those programs will still stay undefined. Strictly technically speaking, the denotations of expressions will now always be defined, because the only source of expression undefinedness in IMP was division by zero. However, for the same reason it is good practice in small-step SOS to have the same type of configurations both to the left and to the right of the transition arrow (\(\rightarrow\)), it is good practice in denotational semantics to work with domains of partial functions instead of ones of total functions. This way, we wouldn’t have to change these later on if we add new expression constructs to the language that yield undefinedness (such as, e.g., recursive functions).

The denotation functions of expressions need to change in order to initiate a “catchable” error when division by zero takes place, and then to propagate it through all the other expression constructs. We only discuss the denotation of division, the other being simpler. The previous denotation function of division \(\llbracket a_1 / a_2 \rrbracket\) was defined as

\[
\llbracket a_1 / a_2 \rrbracket_\sigma = \begin{cases} 
\llbracket a_1 \rrbracket_\sigma \div \llbracket a_2 \rrbracket_\sigma & \text{if } \llbracket a_2 \rrbracket_\sigma \neq 0 \\
\bot & \text{if } \llbracket a_2 \rrbracket_\sigma = 0
\end{cases}
\]

which is not good enough anymore. To catch and propagate the division-by-zero error, we can modify the denotation of division as follows:

\[
\llbracket a_1 / a_2 \rrbracket_\sigma = \begin{cases} 
\llbracket a_1 \rrbracket_\sigma \div \llbracket a_2 \rrbracket_\sigma & \text{if } \llbracket a_1 \rrbracket_\sigma \neq \text{error} \text{ and } \llbracket a_2 \rrbracket_\sigma \neq \text{error} \text{ and } \llbracket a_2 \rrbracket_\sigma \neq 0 \\
\bot & \text{if } \llbracket a_1 \rrbracket_\sigma = \bot \\
\text{error} & \text{if } \llbracket a_1 \rrbracket_\sigma = \text{error} \text{ or } \llbracket a_2 \rrbracket_\sigma = \text{error} \text{ or } \llbracket a_2 \rrbracket_\sigma = 0
\end{cases}
\]

The second case above is necessary, because we want \(\llbracket a_1 / a_2 \rrbracket_\sigma\) to be undefined, and not error, when \(\llbracket a_1 \rrbracket_\sigma = \bot\) and \(\llbracket a_2 \rrbracket_\sigma = 0\).
Like in the previous semantics, the implicit expression errors need to propagate through the statements and halt the program. The denotation function of statements now returns both a state and a flag. The flag tells whether the state resulted from a normal evaluation or whether it is a halting state that needs to be propagated. Here is the new denotation of assignment:

\[
\llbracket x = a; \rrbracket \sigma = \begin{cases} 
(\sigma, \text{halting}) & \text{if } \llbracket a \rrbracket \sigma = \text{error} \\
\bot & \text{if } \sigma(x) = \bot \text{ or } \llbracket a \rrbracket \sigma = \bot \\
(\sigma[\llbracket a \rrbracket \sigma/x], \text{ok}) & \text{if otherwise}
\end{cases}
\]

Above, we chose to halt when \(\llbracket a \rrbracket \sigma = \text{error}\) and \(\sigma(x) = \bot\) (the cases are handled in order). The alternative would be to choose undefined instead of halt (see Exercise 100). Our choice to assume that the expression being assigned is always evaluated no matter whether the assigned variable is declared or not, is consistent with the small-step SOS of assignment in \(\text{IMP}\): first evaluate the expression being assigned step by step, then write the resulting value to the assigned variable if declared; if undeclared then get stuck (Section 3.3.2). The statement sequential composition construct needs to also properly propagate the halting situation:

\[
\llbracket s_1 \; s_2 \rrbracket \sigma = \begin{cases} 
\llbracket s_1 \rrbracket \sigma & \text{if } 2^{\text{nd}}(\llbracket s_1 \rrbracket \sigma) = \text{halting} \\
\llbracket s_2 \rrbracket (1^{\text{st}}(\llbracket s_1 \rrbracket \sigma)) & \text{if } 2^{\text{nd}}(\llbracket s_1 \rrbracket \sigma) = \text{ok} \\
\bot & \text{if } \llbracket s_1 \rrbracket \sigma = \bot
\end{cases}
\]

We also discuss the denotational semantics of loops while \((b) \; s\). It now needs to be the fixed point of a (total) function of the form

\[
\mathcal{F} : (\text{State} \to \text{State} \times \{\text{halting, ok}\}) \to (\text{State} \to \text{State} \times \{\text{halting, ok}\})
\]

The following defines such an \(\mathcal{F}\) which has the right type and captures the information that is added by unrolling the loop once (the cases are handled in order, from top to bottom):

\[
\mathcal{F}(\alpha)(\sigma) = \begin{cases} 
\bot & \text{if } \llbracket b \rrbracket \sigma = \bot \\
(\sigma, \text{halting}) & \text{if } \llbracket b \rrbracket \sigma = \text{error} \\
(\sigma, \text{ok}) & \text{if } \llbracket b \rrbracket \sigma = \text{false} \\
\llbracket s \rrbracket \sigma & \text{if } \llbracket b \rrbracket \sigma = \text{true} \text{ and } 2^{\text{nd}}(\llbracket s \rrbracket \sigma) = \text{halting} \\
\alpha(1^{\text{st}}(\llbracket s \rrbracket \sigma)) & \text{if } \llbracket b \rrbracket \sigma = \text{true} \text{ and } 2^{\text{nd}}(\llbracket s \rrbracket \sigma) = \text{ok}
\end{cases}
\]

After we modify the denotation functions of almost all expression and statement constructs as explained above (except for the denotations of variable lookup, and of built-in integers and Booleans), we have to also modify the denotation of programs to silently discard the halting signal in case its body statement terminated abruptly (the type of the denotation of programs does not change):

\[
\llbracket \text{int.xl}; \; s \rrbracket = 1^{\text{st}}(\llbracket s \rrbracket (x \mapsto 0))
\]

Finally, we can now also give the denotational semantics of “halt;”:

\[
\llbracket \text{halt}; \rrbracket \sigma = (\sigma, \text{halting})
\]

Like for the other semantic extensions in this section, adding the semantics of abrupt termination was easy; the tedious part was to modify the existing semantics to make it aware of abrupt termination.
3.5.4 Adding Dynamic Threads

IMP++ adds threads to IMP, which can be dynamically created and terminated. Like with the previous IMP extensions we keep the syntax and the semantics of threads minimal, and pretend no other features will be added to the language. Recall that as part of our language design experiment, we first only consider thread spawning as a statement, without worrying about thread synchronization. Then, once all the features are put together in the IMP++ language in Section [3.5.6] we will add thread joining synchronization in Section [3.5.7] and, as part of that, will modify thread spawning to be an expression instead of a statement construct, evaluating to the unique identifier of the newly created thread. Therefore, for now we only add a spawn statement construct:

$$\text{Stmt ::= spawn Block}$$

The semantics of spawn $S$ is that the block statement $S$ executes concurrently with the rest of the program, sharing and possibly concurrently modifying the same variables, like threads do. The thread corresponding to $S$ terminates when $S$ terminates and, when that happens, we remove the thread. Like in the previous language extensions, we want programs to terminate normally, no matter whether they make use of threads or not. For example, the programs below create 101 and, respectively, 2 new threads during their execution:

```c
int m, n, s;
int x;
n = 100;
while (m <= n) {
    spawn {s = s + m;}
    m = m + 1;
}
```

Yet, we want the result configurations at the end of the execution to look like before in IMP, that is, like $<\text{skip}, m |\rightarrow 101 & n |\rightarrow 100 & s |\rightarrow 5050>$ and $<\text{skip}, x |\rightarrow 111>$, respectively, in the case of small-step SOS. We grouped the two spawn statements in the second program on purpose, to highlight the need for structural equivalences (after the block is eliminated; this will be discussed shortly, in the context of small-step SOS). Recall that the syntax of IMP’s sequential composition in Section [3.1] (see Figure [3.1]) was deliberately left ambiguous, based on the hypothesis that the semantics of IMP will be given in such a way (left-to-right statement evaluation) that the syntactic ambiguity is irrelevant. Unfortunately, the addition of threads makes the above hard or impossible to achieve modularly in some of the semantic approaches.

Concurrency often implies non-determinism, which is not always desirable. For example, the first program above can also evaluate to a result configuration in which $s$ is 0. This happens when the first spawned thread calculates the sum $s + m$, which is 0, but postpones writing it to $s$ until all the subsequent 100 new threads complete their execution. Similarly, after the execution of the second program above, $x$ can have any of the values 1, 10, 11, 100, 101, 110, 111 (see Exercise [101]).

A language designer or semanticist may find it very useful to execute and analyze programs like above in their semantics. Indeed, the existence of certain behaviors or their lack of, may suggest changes in the syntax and the semantics of the language at an early and therefore cheap design stage. For example, one may decide that one’s language must be race-free on any variable except some special semaphore variables used specifically for synchronization purposes (this particular decision may be too harsh on implementations, though, which may end up having to rely on complex static analysis front-ends or even ignoring it). We make no such difficult decisions in our simple language here, limiting our goal to the bottom of the spectrum of semantic possibilities: we only aim at giving a semantics that faithfully captures all the program behaviors due to spawning threads.

We make the simplifying assumptions that we have a sequentially consistent memory and that variable read and write operations are atomic and thus unaffected by potential races. For example, if $x$ is 0 in a
two-threaded program where one thread is about to write 5 to \( x \) and the other is about to read \( x \), then the only global next steps can be either that the first thread writes 5 to \( x \) or that the second thread reads 0 as the value of \( x \); in other words, we assume that it is impossible for the second thread to read 5 or any other value except 0 as the next small-step step in the program.

**Big-Step SOS**

Big-step SOS and denotational semantics are the semantical approaches which are the most affected by concurrency extensions of the base language. That is because their holistic view of computation makes it hard or impossible to capture the fine-grained execution steps and behavior interleavings that are inherent to concurrent program executions. Consider, for example, a statement of the form \((\text{spawn } S_1) \ S_2\) in some program state \( \sigma \). The only thing we can do in big-step SOS is to evaluate \( S_1 \) and \( S_2 \) in some appropriate states and then to combine their resulting states into a result state. We do not have much room for imagination here: we either evaluate \( S_1 \) in state \( \sigma \) and then \( S_2 \) in the resulting state, or evaluate \( S_2 \) in state \( \sigma \) and then \( S_1 \) in the resulting state. The big-step SOS rule for sequential composition already implies the former case provided that \text{spawn} \( S \) can evaluate to whatever state \( S \) evaluates to, which is true and needs to be considered anyway. Thus, we can formalize the above into the following two big-step SOS rules, which can be regarded as a rather desperate attempt to use big-step SOS for defining concurrency:

\[
\begin{align*}
\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle \\
\langle \text{spawn } s, \sigma \rangle \Downarrow \langle \sigma' \rangle
\end{align*}
\]

(BigStep-Spawn-Arg)

As expected, the two big-step SOS rules above capture only a limited number of the possible concurrent behaviors even for small and simple programs like the ones discussed above. One may try to change the entire big-step SOS definition of IMP to collect in result configurations all possible ways in which the corresponding fragments of program can evaluate. However, in spite of its non-modularity, there seems to be no easy way to combine, for example, the behaviors of \( S_1 \) and of \( S_2 \) into the behaviors of \((\text{spawn } S_1) \ S_2\).

**Type System using Big-Step SOS**

From a typing perspective, \text{spawn} is nothing but a language construct expecting a statement as argument and producing a statement as result. To type programs using \text{spawn} statements we therefore add the following typing rule to the already existing typing rules in Figure 3.10:

\[
\begin{align*}
\Gamma \vdash s : \text{stmt} \\
\Gamma \vdash \text{spawn } s : \text{stmt}
\end{align*}
\]

(BigStepTypeSystem-Spawn)

**Small-Step SOS**

Small-step semantics are more appropriate for concurrency, because they allow a finer-grain view of computation. For example, they allow to say that the next computational step of a statement of the form \((\text{spawn } S_1) \ S_2\) comes either from \( S_1 \) or from \( S_2 \) (which is different from saying that either \( S_1 \) or \( S_2 \) is evaluated next all the way through, like in big-step SOS). Since in this context \text{spawn} \( S_1 \) is already permitted to advance by the small-step SOS rule for sequential composition, the following three small-step SOS rules achieve the desired behavioral non-determinism caused by concurrent threads ... or at least it may look like:
\[ \langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle \] (\texttt{SmallStep-Spawn-Arg})

\[ \langle \text{spawn } s, \sigma \rangle \rightarrow \langle \text{spawn } s', \sigma' \rangle \] (\texttt{SmallStep-Spawn-Skip})

\[ \langle \text{spawn } \{\}, \sigma \rangle \rightarrow \{\}, \sigma \] (\texttt{SmallStep-Spawn-Skip})

\[ \langle s_2, \sigma \rangle \rightarrow \langle s'_2, \sigma' \rangle \] (\texttt{SmallStep-Spawn-Wait})

\[ \langle \text{spawn } s_1, s_2, \sigma \rangle \rightarrow \langle \text{spawn } s_1, s'_2, \sigma' \rangle \] (\texttt{SmallStep-Spawn-Wait})

The rule (\texttt{SmallStep-Spawn-Skip}) cleans up terminated threads.

Unfortunately, the three rules above are not sufficient to capture all the intended behaviors. Consider, for example, the second program at the beginning of Section 3.5.4. That program was intended to have seven different behaviors with respect to the final value of \( x \). Our current small-step SOS misses two of those behaviors, namely those in which \( x \) results in 1 and 100, respectively.

In order for the program to terminate with \( x = 1 \), it needs to start the first new thread, calculate the sum \( x + 1 \) (which is 1), then delay writing it back to \( x \) until after the second and the main threads do their writes of \( x \). However, in order for the main thread to be allowed to execute its assignment statement, the two grouped \text{spawn} statements need to either terminate and become \( \{\} \) so that the rule (\texttt{SmallStep-Seq-Skip}) (see Figure 3.15) applies, or to reduce to only one \text{spawn} statement so that the rule (\texttt{SmallStep-Spawn-Wait}) above applies. Indeed, these are the only two rules which allow access to the second statement in a sequential composition. The first case is not possible, because, as explained, the first newly created thread cannot be terminated. In order for the second case to happen, since the first \text{spawn} statement cannot terminate, the only possibility is for the second \text{spawn} statement to be executed all the way through (which is indeed possible, thanks to the rules (\texttt{SmallStep-Seq-Arg1}) in Figure 3.15 and (\texttt{SmallStep-Spawn-Wait}) above) and then eliminated. To achieve this elimination, we may think of adding a new rule, which appears to be so natural that one may even wonder “how did we miss it in our list above?”:

\[ \langle \text{spawn } s_1, \sigma \rangle \rightarrow \langle \text{spawn } s_1, \sigma' \rangle \] (\texttt{SmallStep-Spawn-Wait})

This rule turns out to be insufficient and, once we fix the semantics properly, it will actually become unnecessary, which is why we did not add it above. Nevertheless, if we add this rule, the resulting small-step SOS can also produce the behavior in which \( x = 1 \) at the end of the execution of the second program at the beginning of Section 3.5.4. However, it is still insufficient to produce the behavior in which \( x = 100 \).

In order to produce a behavior in which \( x = 100 \) when executing the second program, the main thread should first execute its \( x + 100 \) assignment (which evaluates to 100), then let the two child threads do their writes to \( x \), and then write the 100 to \( x \). We have, unfortunately, no rule that allows computations within \( s_2 \) in a sequential composition \( s_1 \ s_2 \) where \( s_1 \) is different from \( \{\} \) or a \text{spawn} statement, as it is our case here. What we want is some generalization of the rule (\texttt{SmallStep-Spawn-Wait}) above which allows computations in \( s_2 \) whenever it is preceded by a sequence of spawns, possibly grouped in possibly nested blocks. On paper definitions, one can do that rather informally by means of some informal side condition saying so. If one needs to be formal, which is a must when one needs to execute the resulting language definitions as we do here, one can define a special sequent saying that a statement only spawns new threads and does nothing else (in the same spirit as the the \( C \backslash \bot \) sequents in Exercise 64), and then use it to generalize the rule (\texttt{SmallStep-Spawn-Wait}) above. However, that would be a rather particular and potentially non-modular solution (what if later on we add agents or other mechanisms for concurrency or grouping?).

Our general solution is to instead enforce the sequential composition of IMP to be structurally associative, using the following structural identity:

\[ x + 1 = x + 1 \]
That means that the small-step SOS reduction rules now apply *modulo* the associativity of sequential composition, that is, that it suffices to find a structurally equivalent representative of a syntactic term which allows a small-step SOS rule to apply. In our case, the program obtained from the original program after eliminating the block surrounding the two `spawn` statements, is structurally equivalent to one whose first statement is the first `spawn` and whose second statement is the sequential composition of the second `spawn` and the assignment of the main thread, and that structurally equivalent program allows all seven desired behaviors, so the original program also allows them. It is important to understand that we cannot avoid enforcing associativity (or, alternatively, the more expensive solution discussed above) by simply parsing the original program so that we start with a right-associative arrangement of the sequentially composed statements. The problem is that right-associativity may be destroyed as the program executes, for example when applying the true/false rules for the `if` statement, so it needs to be dynamically enforced.

Structural identities are not easy to execute and implement, because they can quickly yield an exponential explosion in the number of terms that need to be matched by rules. Since in our particular case we only need the fully right-associative representative of each sequential composition, we can even replace the structural identity above by a small-step SOS rule. The problem with doing that, though, is that the intended computational granularity of the language is significantly modified; for example, the application of a true/false rule for the conditional statement may trigger as many such rearrangement steps as statements in the chosen branch; such rearrangement steps could dominate the total number of steps seen in some computations.

Unfortunately, there is an important syntactic detail that we purposely left out in order to highlight a common problem when using syntactic approaches to operational semantics: it may be possible that terms change their syntactic category during reduction, which may require extending the syntax of some constructs or otherwise require significant changes in the existing semantics of other language constructs. Consider, for example, the rule (\texttt{SmallStep-Spawn-Arg}) above. Syntactically, the rule does not parse, because `spawn` takes a block, not a statement, as argument. We should reduce its applicability only to blocks (i.e., require $s$, $s'$ range over blocks only) in order to obey the declared syntax, but note that if we do so then the rule can only be applied in very particular situations. Indeed, if $s$ is a block \{ $s_1$ \}, then the rule (\texttt{SmallStep-Block}) in Figure 3.15 can be applied to reduce $s$ to $s_1$, but, however, the restricted rule (\texttt{SmallStep-Spawn-Arg}) can only be applied with this premise if $s_1$ is a also block, so $s$ has the form \{ \ldots \}, which is unlikely.

One could argue that the culprit for the above is in fact the semantic rule of blocks, (\texttt{SmallStep-Block}) in Figure 3.15, because it is allowed to change the syntactic category of its code from Block to Stmt. Although there is no SOS restriction saying that the type of the code cannot be changed during the reduction process (in fact this happens quite frequently, e.g., expressions reduce to integers, etc.), let us briefly discuss how the semantics of blocks needs to change in order to reduce blocks to blocks, and what additional problems that generates. First, we would need to propagate the reduction relation through the block construct:

\[
\langle s, \sigma \rangle \rightarrow \langle s', \sigma \rangle
\]

\[
\langle \{ s \}, \sigma \rangle \rightarrow \langle \{ s' \}, \sigma \rangle
\]

Then, once the statement within the block reduces completely, the entire block reduces to the empty block:

\[
\langle \{ \} \rangle, \sigma \rangle \rightarrow \langle \{ \}, \sigma \rangle
\]

While the above may seem a reasonable alternative to the semantics of blocks, it is more involved than our original semantics, slower when we execute it, non-modular (because in order to fix the semantics of a language construct we had to significantly modify the semantics of another language construct which worked
and worse, still does not solve our problem. Indeed, there is no way now to reduce $S_2$ in a fragment of the form $\{ \text{spawn } S_1 \} \ S_2$, because we have to first completely eliminate the block before we move on to reducing $S_2$. The rule ($\text{SmallStep-Spawn-Wait}$) above does not work anymore. More involved changes to the semantics are needed in order to fix the above.

Instead of changing the semantics of existing constructs, let us now investigate more modular approaches to fix the semantics of $\text{spawn}$. One possibility is to structurally enforce $\text{spawn}$ to only take blocks and to only reduce the statements inside the blocks, which can be achieved by changing the rule ($\text{SmallStep-Spawn-Arg}$) above as follows (and keeping the rules ($\text{SmallStep-Spawn-Skip}$) and ($\text{SmallStep-Spawn-Wait}$) unchanged):

$$\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$$

$$\langle \text{spawn } \{ s \}, \sigma \rangle \rightarrow \langle \text{spawn } \{ s' \}, \sigma' \rangle$$

We also have to add one rule to eliminate the block once its enclosed statement completely reduces:

$$\langle \text{spawn } \{ \} , \sigma \rangle \rightarrow \langle \{ \}, \sigma \rangle$$

The last rule above is somewhat artificial, particularly in combination with the rule ($\text{SmallStep-Spawn-Skip}$), which is still necessary. Additionally, both rules above make the connection between $\text{spawn}$ and blocks too tight, indicating that the semantics of $\text{spawn}$ may need to be revisited if we extend the language with other constructs for blocks (e.g., blocks with local variable declarations), or if we extend $\text{spawn}$ to take other statements besides blocks as argument (e.g., function calls).

Another solution, which we prefer here, is to keep the three small-step SOS rules ($\text{SmallStep-Spawn-Arg}$), ($\text{SmallStep-Spawn-Skip}$) and ($\text{SmallStep-Spawn-Wait}$) unchanged, but to extend the syntax of $\text{spawn}$ to take any statements as argument, not only blocks:

$$\text{Stmt ::= } | \text{spawn Stmt}$$

This can be done either by replacing the previous $\text{spawn Block}$ construct with the above or by simply adding the construct above to the already existing grammar of IMP and thus overloading the previous $\text{spawn Block}$ construct. In either case, we should make sure that the original program only uses the syntax $\text{spawn Block}$.

### Denotational Semantics

As already mentioned when we discussed the big-step SOP of $\text{spawn}$ above, big-step SOS and denotational semantics are the semantic approaches which are the most affected by the addition of concurrency to IMP. While big-step SOS was somewhat able to capture some of the non-determinism due to concurrency, unfortunately, denotational semantics cannot do even that easily. The notes on denotational semantics in Section $3.4.3$ mention the use of powerdomains and resumptions when giving concurrent semantics to languages. These are complex denotational semantics topics, which are not easy to use even by experts. Moreover, they yield semantics which are either non-executable at all or very slow. Since the main emphasis of this book is on operational semantics, we do not discuss these advanced topics in this book. Instead, we simply dissolve the $\text{spawn}$ statements, so we can still execute IMP++ programs using denotational semantics:

$$\llbracket \text{spawn } s \rrbracket = \llbracket s \rrbracket$$

Of course, spawning threads is completely useless with our denotational semantics here.
3.5.5 Adding Local Variables

Blocks with local variable declarations are common to many imperative, object-oriented, and functional languages. In IMP++ we follow the common imperative approach where variables can be declared anywhere inside a block, their *scope* being the rest of the block (whatever follows the declaration); in other words, a declared variable is not visible before its declaration or outside the block declaring it. A declaration of a variable that appears in the scope of another declaration of the same variable is said to *shadow* the original one. For example, the values of \( y \) and \( z \) are 1 and 2, respectively, right before the end of the following two IMP++ blocks (none of the variables are visible outside the given blocks):

\[
\begin{align*}
\{ & \text{ int } x, y, z ; \\
& x = 1 ; \\
& y = x ; \\
& \text{ int } x ; \\
& x = 2 ; \\
& z = x ; \}
\end{align*}
\begin{align*}
\{ & \text{ int } x, y, z ; \\
& x = 1 ; \\
& \{ \text{ int } x ; \\
& x = 2 ; \\
& z = x ; \} \\
& y = x ; \}
\end{align*}
\]

As already explained in the preamble of Section 3.5, the introduction of local variables suggests some syntactic and semantic simplifications in the already existing definition of IMP. For example, since local variable declarations generalize the original global variable declarations of IMP, there is no need for the original global declarations. Thus, programs can be just statements. Therefore, we remove the top-level variable declaration and add the following new syntax:

\[
\begin{align*}
\text{Stmt} & ::= \text{ int List}\{Id\} ; \\
\text{Pgm} & ::= \text{Stmt}
\end{align*}
\]

In each of the semantics, we assume that all the previous rules referring to global variable declarations are removed. Moreover, for semantic clarity, we assume that variable declarations can only appear in blocks (a hypothetical parser can reject those programs which do not conform).

An immediate consequence of the language extension and conventions above is that programs now evaluate to empty states. Indeed, since the initial state in which a program is evaluated is empty and since variable declarations are local to the blocks in which they occur, the state obtained after evaluating a program is empty. This makes it somewhat difficult to test this IMP extension. To overcome this problem, one can either add an output statement to the language like in Section 3.5.2 (we will do this in Section 3.5.6), or manually initialize the state with some “global” variables and then use them undeclared in the program.

It would be quite tedious to give semantics directly to the syntactic constructs above. Instead, we are going to propose another construct which is quite common and easy to give semantics to in each of the semantic approaches, and then statically translate the constructs above into the new construct. The new construct has the following syntax:

\[
\begin{align*}
\text{Stmt} & ::= \text{ let Id = AExp in Stmt}
\end{align*}
\]

Its semantics is as expected: the arithmetic expression is first evaluated to an integer, then the declared variable is bound to that integer possibly shadowing an already existing binding of the same variable, then the statement is evaluated in the new state, and finally the environment before the execution of the \text{let} is recovered. The latter step is executed by replacing the value of the variable after the execution of the statement with whatever it was before the \text{let}, possibly undefined. All the other side effects generated by the statement are kept.
Here we propose a simple set of macros which automatically desugar any program using the block and local variable constructs into a program containing only let and the existing IMP constructs:

\[
(s_1 \ s_2) \ s_3 = s_1 \ (s_2 \ s_3) \\
\text{int } x; \ s = \text{int } x; \ \text{let } x=0 \ \text{in } s \\
\text{int } \cdot; \ s = s \\
s \ \text{int } x; = s \\
\{ \text{int } x; \} = \{ \}
\]

The macros above are expected to be iteratively applied in order, from top to bottom, until no macro can be applied anymore. When that happens, there will be no variable declaration left; all of these would have been systematically replaced by the let construct. The first macro enforces right-associativity of sequential composition. This way, any non-terminal variable declaration (i.e., one which is not the last statement in a block) will be followed, via a sequential composition, by the remainder of the block. The second and the third macros iteratively replace each non-terminal variable declaration by a corresponding let statement, while the fourth and the fifth eliminate the remaining (and useless) terminal variable declarations. From here on we assume that these syntactic desugaring macros are applied statically, before any of the semantic rules is applied; this way, the subsequent semantics will only be concerned with giving semantics to let.

**Big-Step SOS**

The big-step SOS rule of let follows quite closely its informal description above:

\[
\frac{\langle a, \sigma \rangle \Downarrow \langle i \rangle \quad \langle s, \sigma[i/x] \rangle \Downarrow \langle \sigma' \rangle}{\langle \text{let } x=a \ \text{in } s, \sigma \rangle \Downarrow \langle \sigma'[\sigma(x)/x] \rangle} \quad \text{ (BIGSTEP-LET)}
\]

In words, the arithmetic expression \(a\) is first evaluated to some integer \(i\). Then the statement \(s\) is evaluated in state \(\sigma[i/x]\), resulting in a state \(\sigma'\). Then we return \(\sigma'[\sigma(x)/x]\) as the result of the let statement, that is, the state \(\sigma'\) in which we update the value of \(x\) to whatever \(x\) was bound to originally in \(\sigma\). We cannot return \(\sigma'\) as the result of the let, because \(\sigma'\) binds \(x\) to some value which is likely different from what \(x\) was bound to in \(\sigma\) (note that \(s\) is allowed to assign to \(x\), although that is not the main problem here). If \(x\) is undefined in \(\sigma\), that is, if \(\sigma(x) = \bot\), then \(x\) is also undefined in \(\sigma'[\sigma(x)/x]\): indeed, recall from Section [2.4.6](#) that \(\sigma'[\bot/x]\) “undefines” \(x\) in \(\sigma'\).

Since programs are now just statements, their big-step SOS simply reduces to that of statements:

\[
\frac{\langle s, \cdot \rangle \Downarrow \langle \sigma \rangle}{\langle s \rangle \Downarrow \langle \sigma \rangle} \quad \text{ (BIGSTEP-PGM)}
\]

Hence, programs are regarded as statements that execute in the empty state. However, since variable accesses in IMP require the variable to be declared and since all variable declarations are translated into let statements, which recover the state in the variable they bind after their execution, we can conclude that \(\sigma\) will always be empty whenever a sequent of the form \(\langle s \rangle \Downarrow \langle \sigma \rangle\) is derivable using the big-step SOS above. The rule above is, therefore, not very useful. All it tells us is that if \(\langle s \rangle \Downarrow \langle \sigma \rangle\) is derivable then \(s\) is a well-formed program which terminates. The idea of reducing the semantics of statement-programs to that of statements in an initial state is general though, and it becomes practical when we add other features to the language (see Section [3.5.6](#)).
Type System using Big-Step SOS

Following the intuitions above, to type programs using let statements we add the following typing rules to the already existing typing rules in Figure 3.10:

\[
\frac{x l \vdash a : \text{int}}{x l, x \vdash \text{let} \ x = a \ \text{in} \ s : \text{stmt}} \quad \text{(BigStepTypeSystem-Let)}
\]

\[
\vdash s : \text{stmt} \quad \vdash s : \text{pgm} \quad \text{(BigStepTypeSystem-Pgm)}
\]

Small-Step SOS

The small-step SOS of let \( x = a \ \text{in} \ s \) can be described in words as follows: first evaluate \( a \) stepwise, until it becomes some integer; then evaluate \( s \) stepwise in a state originally binding \( x \) to the integer to which \( a \) evaluates, but making sure that the value bound to \( x \) is properly updated during each step in the evaluation of \( s \) and it is properly recovered after each step to whatever it was in the environment outside the let (so other potentially interleaved rules taking place outside the let see a consistent state); finally, dissolve the let when its enclosed statement becomes \( \{} \). All these can be achieved with the following three rules, without having to change anything in the already existing small-step SOS of IMP:

\[
\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle \quad \text{(SmallStep-Let-Exp)}
\]

\[
\langle \text{let} \ x = a \ \text{in} \ s, \sigma \rangle \rightarrow \langle \text{let} \ x = a' \ \text{in} \ s, \sigma \rangle \quad \text{(SmallStep-Let-Stmt)}
\]

\[
\langle \text{let} \ x = i \ \text{in} \ s, \sigma \rangle \rightarrow \langle \text{let} \ x = \sigma'(x) \ \text{in} \ s', \sigma'[\sigma(x)/x] \rangle \quad \text{(SmallStep-Let-Done)}
\]

Note that if \( x \) was undeclared before the let then so it stays after each application of the rule (SmallStep-Let-Stmt), because \( \sigma'[\bot/x] \) “undefines” \( \sigma' \) in \( x \) (see Section 2.4.6).

Like in big-step SOS, the semantics of programs (which are now statements) reduces to that of statements. One simple way to achieve that in small-step SOS is to add a rule \( \langle s \rangle \rightarrow \langle s, \cdot \rangle \), in the same spirit as the small-step SOS of IMP in Section 3.3.2 (Figure 3.15). However, like in Exercise 67, one could argue that this approach is wasteful, since one does not want to spend a step only to initialize the empty state (this can be regarded as poor style). For demonstration purposes and for the sake of a semantic variation, we here prefer a non-wasteful small-step SOS rule of programs:

\[
\langle s, \cdot \rangle \rightarrow \langle s', \sigma \rangle \quad \text{(SmallStep-Pgm)}
\]

One could still argue that the rule above is not perfect, because the configuration \( \langle \{} \rangle \) is frozen; thus, while any other (terminating) program eventually reduces to a configuration of the form \( \langle \{} \cdot \rangle \), \( \{} \) itself does not. To address this non-uniformity problem, one can add a rule \( \langle \{} \rangle \rightarrow \langle \{} \cdot \rangle \); this wastes a step, indeed, but this case when the entire program is just \( \{} \) is expected to be very uncommon. A conceptually cleaner alternative is to replace the rule (SmallStep-Pgm) above with a structural identity \( \langle s \rangle \equiv \langle s, \cdot \rangle \) identifying each program configuration with a statement configuration holding an empty state. This can be easily achieved in Maude using an equation, but it can be harder to achieve in other rewrite systems providing support and semantics only for rewrite rules but not for equations.
Denotational Semantics

The denotational semantics of the \texttt{let} construct is very compact and elegant:

\[
\llbracket \texttt{let } x = a \texttt{ in } s \rrbracket \sigma = (\llbracket s \rrbracket(\sigma[\llbracket a \rrbracket(\sigma/x)]))(\sigma(x)/x)
\]  

(DENOTATIONAL-LET)

Like in the other semantics above, this works because $\sigma'[\bot/x]$ “undefines” $x$ in $\sigma'$ (see Section 2.4.6).

3.5.6 Putting Them All Together: First Attempt

In this section and the next we analyze the modularity of the various semantic approaches discussed so far by defining the IMP++ language, which puts together all the language features discussed so far plus more. Recall from the preamble of Section 3.5 that, as part of our language design experiment, we first consider spawn to be a statement construct with no explicit thread synchronization constructs. This is also what we did in Section 3.5.4. Our objective in this section is therefore to “make everything work” under this assumption, and then, in Section 3.5.7, we change spawn into an expression construct that evaluates to a unique thread identifier, and add join thread synchronization. Therefore, the IMP++ language discussed in this section removes from IMP the global variable declarations and adds the following constructs:

\[
AExp ::= ++Id \\
| \text{read()}
\]

\[
Stmt ::= \text{print}(AExp); \\
| \text{halt}; \\
| \text{spawn Block} \\
| \text{int List}\{Id\};
\]

\[
Pgm ::= Stmt
\]

We consider the semantics of these constructs adopted in the previous sections.

To make the design of IMP++ more permissive in what regards its possible implementations, we shall opt for maximum non-determinism whenever such design choices can be made. For example, in the case of division expressions $a_1/a_2$, we want to capture all possible behaviors (recall that division is non-deterministic) due to the possibly interleaved evaluations of $a_1$ and $a_2$, including all possible abruptly terminated behaviors generated when $a_2$ evaluates to 0. In particular, we want to also capture those behaviors where $a_1$ is not completely evaluated. The rationale for this language design decision is that we want to allow maximum flexibility to implementations of IMP++; for example, some implementations may choose to evaluate $a_1$ and $a_2$ in two concurrent threads and to stop with abrupt termination as soon as the thread evaluating $a_2$ yields 0.

Somewhat surprisingly, when adding several new features together to a language, it is not always sufficient to simply apply all the global, non-modular changes that are required for each feature in isolation. We sometimes have to additionally consider the semantic implications of the various combinations of features. For example, the addition of side-effects in combination with division-by-zero abrupt termination requires the addition of new rules to catch specific new behaviors due to this particular combination. Indeed, the evaluation of $a_1/a_2$, for example, may abruptly terminate with the current state precisely when $a_2$ is evaluated to zero, but it can also terminate with the state obtained after evaluating $a_1$ or parts of $a_1$, as discussed above.

Also, a language design decision needs to be made in what regards the state of abruptly terminated programs. One option is to simply enclose the local state when the abrupt termination flag was issued. This option is particularly useful for debugging. However, as argued in Section 3.5.3, we want abruptly terminated programs to behave the same way as the normally terminated programs. Since normally terminated programs now empty the state after their execution, we will give the semantics of abruptly terminated programs to also...
empty the state. Whenever easily possible, we will give the semantics of abruptly terminated statements to return after their evaluation a state binding precisely the same variables as before their evaluation.

A design decision also needs to be made in what regards the interaction between abrupt termination and threads. We choose that abrupt termination applies to the entire program, no matter whether it is issued by the main program or by a spawned thread. An alternative would be that abrupt termination only applies to the spawned thread if issued by a spawned thread, or to the entire program if issued by the main program. Yet another alternative is to consider the main program as an ordinary thread, and an abrupt termination issued by the main program to only stop that thread, allowing the other spawned threads to continue their executions.

Finally, there is an interesting aspect regarding the interaction between blocks and threads. In conventional programming languages, spawned threads continue to execute concurrently with the rest of the program regardless of whether the language construct which generated them completed its execution or not. For example, if function \( f() \) spawns a thread and then immediately returns 1, then the expression \( f() + f() \) evaluates to 2 and the two spawned threads continue to execute concurrently with the rest of the program. We do not have functions in \( \text{IMP}++ \), but we still want the spawned threads to continue to execute concurrently with the rest of the program even after the completion of the block within which the spawn statements were executed. For example, we would like the \( \text{IMP}++ \) program (its \texttt{let}-desugared variant is shown to the right)

```plaintext
{ int x ;
 { int y ;
 spawn { x := x + 1 ; } 
}
spawn { x := x + 10 ; }
print(x) ; }

let x = 0 in ( 
 { let y = 0 in 
 spawn { x := x + 1 ; } 
}
spawn { x := x + 10 ; }
print(x) ; )
```

to manifest four behaviors, where \( x \) is 0, 1, 10, and 11, and not only two (where \( x \) is 1, 11) as it would be the case if the first spawn statement were not allowed to transcend its surrounding block.

Below we discuss, for each semantic approach, the changes that we have to apply to the semantics of \( \text{IMP} \) in order to extend it to \( \text{IMP}++ \), highlighting changes that cannot be mechanically derived from the changes required by each of \( \text{IMP}++ \)'s features when considered in isolation.

**Big-Step SOS**

To accommodate the side effects generated by variable increment on the state and by \texttt{read()} on the input buffer, and the possible abrupt termination generated by division-by-zero, the arithmetic expression sequents need to change from \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) to \( \langle a, \sigma, \omega \rangle \Downarrow \langle i, \sigma', \omega' \rangle \) for normal termination and to \( \langle a, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma', \omega' \rangle \) for abrupt termination, and similarly for Boolean expressions, where \( \sigma, \omega \) and \( \sigma', \omega' \) are the states and input buffers before and after the evaluation of \( a \), respectively. Also, the original elegant big-step SOS rules for expressions need to change to take into account the new configurations, the various side effects, and the abrupt termination due to division-by-zero. For example, the \( \text{IMP} \) big-step SOS rule for division in Section 3.2.2 namely

\[
\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle
\]

\[
\langle a_1 / a_2, \sigma \rangle \Downarrow \langle i_1 +_{\text{err}} i_2 \rangle \quad \text{if } i_2 \neq 0
\]
We also have to include big-step SOS rules for input declarations like those in Section 3.5.5, and for dynamic threads like those in Section 3.5.4, but also modified termination are properly propagated, like we did in Sections 3.5.1 and 3.5.3 (but for the new configurations). For statements need to change to accommodate the new sequents, making sure that side effects and abrupt termination abruptly; here 

\[ \text{changes into the following six rules:} \]

\[
\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \quad \langle a_2, \sigma_1, \omega_1 \rangle \Downarrow \langle i_2, \sigma_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle i_1 / i_2, \sigma_2, \omega_2 \rangle} \quad \text{if } i_2 \neq 0
\]

\[
\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \quad \langle a_2, \sigma_1, \omega_1 \rangle \Downarrow \langle \text{error}, \sigma_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_2, \omega_2 \rangle}
\]

\[
\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_1, \omega_1 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_1, \omega_1 \rangle}
\]

\[
\frac{\langle a_1, \sigma_2, \omega_2 \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \quad \langle a_2, \sigma, \omega \rangle \Downarrow \langle i_2, \sigma_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle i_1 / i_2, \sigma_1, \omega_1 \rangle} \quad \text{if } i_2 \neq 0
\]

\[
\frac{\langle a_1, \sigma_2, \omega_2 \rangle \Downarrow \langle \text{error}, \sigma_1, \omega_1 \rangle \quad \langle a_2, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_1, \omega_1 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_2, \omega_2 \rangle}
\]

Like for the individual features, rules like the above attempt to capture the intended nondeterministic evaluation strategy of the arithmetic operators, but they end up capturing only the non-deterministic choice semantics. In the case of division, we also have to add the rule for abrupt termination in the case of a division-by-zero, like the rule (bigSTEP-Div-By-Zero) in Section 3.5.3. However, since we want to capture all the non-deterministic behaviors that big-step SOS can detect, we actually need three such rules:

\[
\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \quad \langle a_2, \sigma_1, \omega_1 \rangle \Downarrow \langle 0, \sigma_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_2, \omega_2 \rangle}
\]

\[
\frac{\langle a_1, \sigma_2, \omega_2 \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \quad \langle a_2, \sigma, \omega \rangle \Downarrow \langle 0, \sigma_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_2, \omega_2 \rangle}
\]

\[
\frac{\langle a_2, \sigma, \omega \rangle \Downarrow \langle 0, \sigma_2, \omega_2 \rangle}{\langle a_1 / a_2, \sigma, \omega \rangle \Downarrow \langle \text{error}, \sigma_2, \omega_2 \rangle}
\]

The big-step SOS sequents for statements \((s, \sigma) \Downarrow (\sigma')\) also need to change, to hold both the input/output buffers and the termination flag. We use sequents of the form \((s, \sigma, \omega_{in}) \Downarrow (\sigma', \omega'_{in}, \omega_{out})\) for the case when \(s\) terminates normally and sequents of the form \((s, \sigma, \omega_{in}) \Downarrow (\text{halting}, \sigma', \omega'_{in}, \omega_{out})\) for the case when \(s\) terminates abruptly; here \(\omega_{in}, \omega'_{in} \in \text{Buffer}\) are the input buffers before and, respectively, after the evaluation of statement \(s\), and \(\omega_{out} \in \text{Buffer}\) is the output produced during the evaluation of \(s\). All the big-step SOS rules for statements need to change to accommodate the new sequents, making sure that side effects and abrupt termination are properly propagated, like we did in Sections 3.5.1 and 3.5.3 (but for the new configurations). We also have to include big-step SOS rules for input/output like those in Section 3.5.2, for local variable declarations like those in Section 3.5.5 and for dynamic threads like those in Section 3.5.4 but also modified
to work with the new configurations and to propagate abrupt termination. Recall from the preamble of this section that we want abruptly terminated programs to terminate similarly to the normal programs, that is, with an empty state in the configuration. This can be easily achieved in the rule for programs by simply emptying the state when the program statement terminates abruptly. However, in the case of big-step SOS it is relatively easy to ensure a stronger property, namely that each statement leaves the state in a consistent shape after its evaluation, no matter whether that is abruptly terminated or not. All we have to do is to also clean up the state when the halting signal is propagated through the let construct. For clarity, we show both big-step SOS rules for let:

\[
\begin{align*}
\langle a, \sigma', \omega_{in} \rangle \Downarrow (i, \sigma'[i/x], \omega'_{in}, \omega_{out}) & \quad \langle s, \sigma'[x/i], \omega'_{in}, \omega_{out} \rangle \Downarrow (\sigma'', \omega'_{in}, \omega_{out}) \\
\langle a, \sigma, \omega_{in} \rangle & \Downarrow \langle i, a, \sigma, \omega_{in} \rangle \\
\langle s, \sigma, \omega_{in} \rangle & \Downarrow \langle \sigma'[x/i], \omega'_{in}, \omega_{out} \rangle
\end{align*}
\]

Recall from Section 2.4.6 that \(\sigma''[\bot/x]\) “undefines” \(x\) in \(\sigma''\).

Finally, the big-step SOS sequents and rules for programs also have to change, to take into account the fact that programs are now just statements like in Section 3.5.5.2 that they take an input and that they yield both the unconsumed input and an output like in Section 3.5.2.2 and that programs manifest normal termination behavior no matter whether their corresponding statement terminates normally or not:

\[
\begin{align*}
\langle s, xl \mapsto \rightarrow 0, \omega_{in} \rangle & \Downarrow \langle \sigma, \omega_{in}, \omega_{out} \rangle \\
\langle s, \omega_{in} \rangle & \Downarrow \langle \sigma, \omega_{in}, \omega_{out} \rangle
\end{align*}
\]

Unfortunately, as seen above, all the configurations, sequents and rules of IMP extended with any one of the features of IMP++ had to change again when we added all the features together. This highlights, again, the poor modularity of big-step SOS. But, even accepting the poor modularity of big-step SOS, do we at least get all the behaviors expressible in a big-step SOS style by simply putting together and adjusting accordingly all the rules of the individual features? Unfortunately, not. Recall from the preamble of Section 3.5.6.6 that we want spawned threads to execute concurrently with the rest of the program also after their surrounding blocks complete. In other words, we would like to also capture behaviors of, e.g., \(\langle \text{let } x = a \text{ in } s_1 \rangle s_2\), where \(a\) is first evaluated, then \(s_2\), and then \(s_1\). We can easily add a big-step SOS rule to capture this particular situation, but is that enough? Of course not, because there are many other similar patterns in which we would like to allow the evaluation of \(s_2\) before the preceding statement completes its evaluation. For example, one can replace \(\text{spawn } s_1\) above by another \(\text{let}\) holding a spawn statement, or by \(\text{spawn } s'_1\text{ spawn } s'_2\), or by combinations of such patterns. A tenacious reader could probably find some complicated way to allow all these behaviors. However, it is fair to say that big-step SOS has simply not been conceived to deal with concurrent languages, and can only partially deal with non-determinism.

**Type System using Big-Step SOS**

The IMP++ type system is quite simple and modular. We simply put together all the typing rules of the individual language features discussed in Sections 3.5.1, 3.5.2, 3.5.3, 3.5.4 and 3.5.5.
Small-Step SOS

It is conceptually easy, though not entirely mechanical, to combine the ideas and changes to the original IMP small-step SOS discussed in Sections 3.5.1, 3.5.2, and 3.5.3, to obtain the small-step SOS rules of IMP++.

The arithmetic expression sequents need to change from \( \langle a, \sigma \rangle \to \langle a', \sigma' \rangle \) to \( \langle a, \sigma, \omega \rangle \to \langle a', \sigma', \omega' \rangle \) for normal steps and to \( \langle a, \sigma, \omega \rangle \to \text{error}, \sigma', \omega' \rangle \) for halting steps, and similarly for Boolean expressions, where \( \sigma, \omega \) and \( \sigma', \omega' \) are the states and input buffers before and after the small-step applied to \( a \), respectively. Also, the original small-step SOS rules for expressions need to change to take into account the new configurations, the various side effects, and the abrupt termination due to division-by-zero. For example, here are the new rules for division:

\[
\begin{align*}
\langle a_1, \sigma, \omega_{in} \rangle &\to \langle a'_1, \sigma', \omega'_{in} \rangle \\
\langle a_1 / a_2, \sigma, \omega_{in} \rangle &\to \langle a'_1 / a_2, \sigma', \omega'_{in} \rangle \\
\langle a_1, \sigma, \omega_{in} \rangle &\to \text{error}, \sigma', \omega'_{in} \\
\langle a_1 / a_2, \sigma, \omega_{in} \rangle &\to \text{error}, \sigma', \omega'_{in} \\
\langle a_2, \sigma, \omega_{in} \rangle &\to \langle a'_2, \sigma', \omega'_{in} \rangle \\
\langle a_1 / a_2, \sigma, \omega_{in} \rangle &\to \langle a'_1 / a_2, \sigma', \omega'_{in} \rangle \\
\langle a_2, \sigma, \omega_{in} \rangle &\to \text{error}, \sigma', \omega'_{in} \\
\langle a_1 / a_2, \sigma, \omega_{in} \rangle &\to \text{error}, \sigma', \omega'_{in} \\
\langle i_1 / i_2, \sigma, \omega_{in} \rangle &\to \langle i_1 /_0 i_2, \sigma', \omega'_{in} \rangle \quad \text{if} \ i_2 \neq 0 \\
\langle a_1 / 0, \sigma, \omega_{in} \rangle &\to \text{error}, \sigma, \omega_{in}
\end{align*}
\]

Note that the last rule above does not require the full evaluation of \( a_1 \) in order to flag abrupt termination. This aspect was irrelevant when we added abrupt termination in isolation to IMP in Section 3.5.3 because expressions did not have side effects there. However, since expressions can now modify both the state (via variable increment) and the input buffer (via \texttt{read()}), the rule above captures more behaviors than a rule replacing \( a_1 \) by an integer \( i_1 \), which would be obtained by mechanically translating the corresponding rule from Section 3.5.3.

The small-step SOS rules of statements and programs result from those of the individual features discussed in Sections 3.5.1, 3.5.2, and 3.5.3 by modifying the configurations to include all the new semantic components. Sequents \( \langle s, \sigma \rangle \to \langle s', \sigma' \rangle \) change into sequents \( \langle s, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle s', \sigma', \omega'_{in}, \omega'_{out} \rangle \) and \( \langle s, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle \text{halting}, \sigma', \omega'_{in}, \omega'_{out} \rangle \) for normal and for halting steps, respectively, where \( \sigma, \omega_{in}, \omega_{out} \) and \( \sigma', \omega'_{in}, \omega'_{out} \) are the states, input buffers and output buffers before and after the small-step applied to \( s \), respectively. The only rule that deviates from the expected pattern is the rule that propagates the halting signal through the \texttt{let} construct. Like in the case of big-step SOS discussed above, we can do it in such a way that the state is consistently cleaned up after each \texttt{let}, regardless of whether its body statement terminated.
abruptly or not. Here are all three small-step SOS rules for `let`:

\[
\langle s, \sigma[i/x], \omega_{in}, \omega_{out} \rangle \rightarrow \langle s', \sigma', \omega'_{in}, \omega'_{out} \rangle
\]

\[
\langle \text{let } x = i \text{ in } s, \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle \text{let } x = \sigma(x) \text{ in } s', \sigma'[\sigma(x)/x], \omega'_{in}, \omega'_{out} \rangle
\]

\[
\langle \text{let } x = i \text{ in } [], \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle [], \sigma, \omega_{in}, \omega_{out} \rangle
\]

\[
\langle s, \sigma[i/x], \omega_{in}, \omega_{out} \rangle \rightarrow \langle \text{halting}, \sigma', \omega'_{in}, \omega'_{out} \rangle
\]

\[
\langle \text{let } x = i \text{ in } s, \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle \text{halting}, \sigma'[\sigma(x)/x], \omega'_{in}, \omega'_{out} \rangle
\]

Recall again from Section 2.4.6 that \(\sigma'[\perp/x]\) "undefines" \(x\) in \(\sigma'\).

Even though strictly speaking all the small-step SOS rules above are different from their corresponding rules in the IMP extension introducing only the feature they define, they are somewhat mechanically derivable. In fact, MSOS (see Section 3.6) formalizes and mechanizes this process. The main question is then whether these new small-step SOS rules indeed capture the intended semantics of IMP++. Unfortunately, like in the case of big-step SOS, they fail to capture the intended semantics of `spawn`. Indeed, since the `let` construct does not dissolve itself until its body statement becomes `[]`, statements of the form `(let x = i in spawn s1) s2` will never allow reductions in \(s2\), thus limiting the concurrency of spawn statements to their defining blocks.

There are several ways to address the problem above, each with its advantages and limitations. One possibility is to attempt to syntactically detect all situations in which a statement allows a subsequent statement to execute, that is, all situations in which the former can only perform spawn steps. Like in Section 3.3.2 where we introduced special configurations of the form \(C\) (following Hennessy [32]) called “terminated configurations”, we can introduce special “parallel configurations” \(C\|\) stating that \(C\) can only spawn statements or discard terminated spawned statements, that is, \(C\)’s statement can be executed in parallel with subsequent statements. Assuming such \(C\|\) special configurations, we can remove the existing small-step SOS rule corresponding to rule (SmallStep-Spawn-Wait) in Section 3.5.4 and add instead the following rule:

\[
\langle s_1, \sigma, \omega_{in}, \omega_{out} \rangle \| \langle s_2, \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle s'_2, \sigma', \omega'_{in}, \omega'_{out} \rangle
\]

\[
\langle s_1 \| s_2, \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle s_1, s'_2, \sigma', \omega'_{in}, \omega'_{out} \rangle
\]

Alternatively, one can define \(\|\) as a predicate only taking a statement instead of a configuration, and then move the \(\|\) sequent out from the rule premise into a side condition. Nevertheless, the \(\|\) configurations or predicates need to be defined such that they include only statements of the form `spawn s` and `spawn s` `spawn s'` and `let x = i in spawn s` and combinations of such statements. The idea is that such statements can safely be executed in parallel with what follows them. For example, a statement of the form `let x = a in spawn s` where \(a\) is not a value does not fall into this pattern. Exercise 116 further explores this approach.

One problem with the approach above is that it is quite non-modular. Indeed, the definition of \(\|\) is strictly dependent upon the current language syntax. Adding or removing syntax to the language will require us to also revisit the definition of \(\|\). Another and bigger problem with this approach is that it does not seem to work for other concurrent language constructs. For example, in many languages including our extension of IMP++ in Section 3.5.7 the creation of a thread is an expression (and not a statement) construct, returning to the calling context the new thread identifier as a value. The calling context can continue its execution using the returned value in parallel with the spawned thread. In our context, for example, if `spawn s` returned an integer value, we would have to allow expressions of the form `spawn \{ x = x + 1; \} <= spawn \{ x = x + 10; \} + x` and be able to execute the two threads concurrently with the lookup for \(x\) and the evaluation of the `<=` Boolean expression. It would be quite hard to adapt the approach above to work with such common concurrent constructs which both return a value to the calling context and execute their code in parallel with the context.
Another way to address the loss of concurrent behaviors due to the syntactic constraints imposed by \texttt{let} on the \texttt{spawn} statements in its body, is to eliminate the \texttt{let} as soon as it is semantically unnecessary. Indeed, once the expression \( a \) is evaluated in a construct \texttt{let} \( x = a \in s \), the \texttt{let} is semantically unnecessary. The only reason we kept it was because it offered us an elegant syntactic means to keep track of the execution contexts both for its body statement and for the outside environment. Unfortunately, it is now precisely this “elegant syntactic means” that inhibits the intended concurrency of the \texttt{spawn} statement. One way to eliminate the semantically unnecessary \texttt{let} statements is to try to add a small-step SOS rule of the form:

\[
\langle \texttt{let} \ x = i \in s, \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle s \ x = \sigma(x);, \sigma[i/x], \omega_{in}, \omega_{out} \rangle
\]

The idea here is to reduce the \texttt{let} statement to its body statement in a properly updated state, making sure that the state is recovered after the execution of the body statement. The infusion of the assignment statement “\( x = \sigma(x); \)” is a bit unorthodox, because \( \sigma(x) \) can also be undefined; it works in our case here because we allow state updates of the form \( \sigma[ \bot/x] \), which have the effect to undefine \( x \) in \( \sigma \) (see Section \ref{sec:2.4.6}), and the assignment rule generates such a state update. This trick is not crucial for our point here; if one does not like it, then one can split the rule above in two cases, one where \( \sigma(x) \) is defined and one where it is undefined, and then add a special syntactic statement construct to undefine \( x \) in the second case. The real problem with this \texttt{let}-elimination approach is that it is simply wrong when we also have \texttt{spawn} statements in the language. For example, if \( s \) is a \texttt{spawn} statement then the \texttt{let} reduces to the \texttt{spawn} statement followed by the assignment to \( x \); the small-step SOS rule for \texttt{spawn} allowing the spawned statement to be executed in parallel with its subsequent statement then kicks in and allows the assignment to \( x \) to be possibly evaluated before the spawned statement, thus resulting in a wrong behavior: the spawned statement should only see the \( x \) bound by its \texttt{let} construct, not the outside \( x \) (possibly undefined). A correct way to eliminate the \texttt{let} construct is to rename the bound variable into a fresh variable visible only to \texttt{let}'s body:

\[
\langle \texttt{let} \ x = i \in s, \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle s[x'/x], \sigma[i/x'], \omega_{in}, \omega_{out} \rangle \text{ if } x' \text{ is a fresh variable}
\]

Besides having to provide and maintain (as the language changes) a substitution operation and then having to pay linear or worse complexity each time a \texttt{let} is eliminated, and besides creating potentially unbounded garbage bindings in the state (e.g., the \texttt{let} statement can be inside a loop), the solution above appears to only solve our particular problem with our particular combination of \texttt{let} and \texttt{spawn}. It still does not seem to offer us a general solution for dealing with arbitrary constructs for concurrency, in particular with \texttt{spawn} constructs that evaluate to a value which is immediately returned to the calling context, as described a few paragraphs above. For example, while it allows for renaming the variable \( x \) into a fresh variable when the expression \texttt{spawn} \{ \( x = x + 1; \) \} \texttt{<=} \texttt{spawn} \{ \( x = x + 10; \) \} + \( x \) appears inside a \texttt{let} \( x = i \in ... \) construct, we still have no clear way to evaluate both \texttt{spawn} expressions and the expression containing them concurrently.

The correct solution to deal with concurrency in small-step SOS is to place all the concurrent threads or processes in a syntactic “soup” where any particular thread or process, as well as any communicating subgroups of them, can be picked and advanced one step. Since in our IMP++ language we want all threads to execute concurrently without any syntactic restrictions, we have to place all of them in some top-level “soup”, making sure that each of them is unambiguously provided its correct execution environment. For example, if a thread was spawned from inside a \texttt{let} statement, then it should correctly see precisely the execution environment available at the place where it was spawned, possibly interacting with other threads seeing the same environment; it should not wrongly interfere with other threads happening to have had variables with the same names in their creation environments. This can be done either by using a substitution

\footnote{See Section \ref{sec:4.5.3}}
like above to guarantee that each bound variable is distinct, or by splitting the current state mapping variable identifiers to values into an environment mapping identifiers to memory locations and a store (or memory, or heap) mapping locations to values. In the latter case, each spawned thread would be packed together with its creation environment and all threads would share the store. The environment-store approach is the one that we will follow shortly in Section 3.5.7, so we do not insist on it here.

The morale of the discussion above is that putting features together in a language defined using small-step SOS is a highly non-trivial matter even when the language is trivial, like our IMP++. On the one hand, one has to non-modularly modify the configurations to hold all the required semantic data of all the features and then to modify all the rules to make sure that all these data are propagated and updated appropriately by each language construct. Addressing this problem is the main motivation of the MSOS approach discussed in Section 3.6. One the other hand, the desired feature interactions can require quite subtle changes to the semantics, which are sometimes hard to achieve purely syntactically. One cannot avoid the feeling that syntax is sometimes just too rigid, particularly when concurrency is concerned. This is actually one of the major motivations for the chemical abstract machine computational and semantic model discussed in Section 3.8.

**Denotational Semantics**

In order to accommodate all the semantic data needed by all the features, the denotation functions will now have the following types:

\[
\llbracket \cdot \rrbracket : \text{AExp} \to (\text{State} \times \text{Buffer} \to \text{Int} \cup \{\text{error}\} \times \text{State} \times \text{Buffer})
\]

\[
\llbracket \cdot \rrbracket : \text{BExp} \to (\text{State} \times \text{Buffer} \to \text{Bool} \cup \{\text{error}\} \times \text{State} \times \text{Buffer})
\]

\[
\llbracket \cdot \rrbracket : \text{Stmt} \to (\text{State} \times \text{Buffer} \to \text{State} \times \text{Buffer} \times \text{Buffer} \times \{\text{halting, ok}\})
\]

Moreover, since programs are now statements and since we want their denotation to only take an input buffer and to return a state, the remainder of the input buffer and the output buffer (recall that we deliberately discard the halting status), we replace the original denotation function of programs with the following (we use a different name, to distinguish it from the last denotation function above):

\[
\llbracket pgm \rrbracket : \text{Pgm} \to (\text{Buffer} \to \text{State} \times \text{Buffer} \times \text{Buffer})
\]

For example, the denotation of division becomes

\[
\llbracket a_1 / a_2 \rrbracket = \begin{cases} 
(1^{st}(a_1) +_{\text{mod}} 1^{st}(a_2), \ 2^{nd}(a_2), \ 3^{rd}(a_2)) & \text{if } 1^{st}(a_1) \neq \text{error} \\
\text{error} & \text{and } 1^{st}(a_2) \neq \text{error} \\
\text{error} & \text{and } 1^{st}(a_2) = 0 \\
\text{error} & \text{if } 2^{nd}(a_2) = \text{error} \\
\text{error} & \text{or } 1^{st}(a_2) = 0 \\
\end{cases}
\]

where \( a_1 = \llbracket a_1 \rrbracket \) and \( a_2 = \llbracket a_2 \rrbracket (2^{nd}(a_1), 3^{rd}(a_1)) \), the denotation of sequential composition becomes

\[
\llbracket s_1 \ s_2 \rrbracket = \begin{cases} 
(1^{st}(a_2), \ 2^{nd}(a_2), \ 3^{rd}(a_1) : 3^{rd}(a_2), \ 4^{th}(a_2)) & \text{if } 4^{th}(a_1) = \text{ok} \\
\text{error} & \text{if } 4^{th}(a_1) = \text{halting} \\
\end{cases}
\]

where \( a_1 = \llbracket s_1 \rrbracket \) and \( a_2 = \llbracket s_2 \rrbracket (1^{st}(a_1), \ 2^{nd}(a_1)) \), the denotational semantics of while loops while \((b)\)s become the fixed-points of total functions

\[
\mathcal{F} : (\text{State} \times \text{Buffer} \to \text{State} \times \text{Buffer} \times \text{Buffer} \times \{\text{halting, ok}\}) \\
\rightarrow (\text{State} \times \text{Buffer} \to \text{State} \times \text{Buffer} \times \text{Buffer} \times \{\text{halting, ok}\})
\]

203
defined as

\[ F(\alpha)(\pi) = \begin{cases} 
(1^{\text{st}}(\text{arg}_1), 2^{\text{nd}}(\text{arg}_3), 3^{\text{rd}}(\text{arg}_2) : 3^{\text{rd}}(\text{arg}_3), 4^{\text{th}}(\text{arg}_3)) & \text{if } 1^{\text{st}}(\text{arg}_1) = \text{true} \\
 & \text{and } 4^{\text{th}}(\text{arg}_2) = \text{ok} \\
(2^{\text{nd}}(\text{arg}_1), 3^{\text{rd}}(\text{arg}_1), \epsilon, \text{ok}) & \text{if } 1^{\text{st}}(\text{arg}_1) = \text{false} \\
(2^{\text{nd}}(\text{arg}_1), 3^{\text{rd}}(\text{arg}_1), \epsilon, \text{halting}) & \text{if } 1^{\text{st}}(\text{arg}_1) = \text{error}
\end{cases} \]

where \( \text{arg}_1 = \llbracket b \rrbracket \pi, \text{arg}_2 = \llbracket s \rrbracket (2^{\text{nd}}(\text{arg}_1), 3^{\text{rd}}(\text{arg}_1)), \) and \( \text{arg}_3 = \alpha(1^{\text{st}}(\text{arg}_2), 2^{\text{nd}}(\text{arg}_2)), \) and the denotation of programs is defined as the function

\[ \llbracket s \rrbracket_{\text{pgm}}(\omega) = (1^{\text{st}}(\text{arg}), 2^{\text{nd}}(\text{arg}), 3^{\text{rd}}(\text{arg})) \]

where \( \text{arg} = \llbracket s \rrbracket (\cdot, \omega), \) that is, the program statement \( s \) with input buffer \( \omega \) is evaluated in the empty state “\( \cdot \)” and the input buffer \( \omega \), and the resulting halting flag is discarded.

Once all the changes above are applied to correctly handle the semantic requirements of the various features of \( \text{IMP}^+ \), the denotational semantics of those features is relatively easy, basically a simple adaptation of their individual denotational semantics in Sections 3.5.1, 3.5.2, 3.5.3, 3.5.4, and 3.5.5. We let their precise definitions as an exercise to the reader (see Exercise 119). Note, however, that the resulting denotational semantics of \( \text{IMP}^+ \) is still non-deterministic and non-concurrent. Because of this accepted limitation, we do not worry about the loss of concurrent behaviors due to the interaction between \text{spawn} and \text{let}.

### 3.5.7 Putting Them All Together: Second Attempt

...

### 3.5.8 Notes

The first to pinpoint the limitations of plain SOS and denotational semantics when defining non-trivial languages were the inventors of alternative semantic frameworks, such as Berry and Boudol [8, 9] who proposed the chemical abstract machine model (see Section 3.8), Felleisen and his collaborators [25, 88] who proposed evaluation contexts (see Section 3.7), and Mosses and his collaborators [51, 52, 53] who proposed the modular SOS approach (see Section 3.6). Among these, Mosses is perhaps the one who most vehemently criticized the lack of modularity of plain SOS, bringing as evidence natural features like the ones we proposed for \( \text{IMP}^+ \), which require the structure of configurations and implicitly the existing rules to change no matter whether there is any semantic interaction or not between the new and the old features.

The lack of modularity of SOS was visible even in Plotkin’s original notes [60, 61], where he had to modify the definition of simple arithmetic expressions several times as his initial language evolved. Hennessy also makes it even more visible in his book [32]. Each time he adds a new feature, he also has to change the configurations and the entire existing semantics, similarly to each of our IMP extensions in this section. However, the lack of modularity of language definitional frameworks was not perceived as a major problem until late 1990es, partly because there were few attempts to give complete and rigorous semantics to real programming languages. Hennessy actually used each language extension as a pedagogical opportunity to teach the reader what new semantic components the feature needs and how and where those are located in each sequent. Note also that Hennessy’s languages were rather simple and pure. His imperative language, called \text{WhileL}, was actually simpler even than our IMP (\text{WhileL} had no global variable declarations). Hennessy’s
approach was somewhat different from ours, namely he defined a series of different paradigmatic languages, each highlighting certain semantic aspects in a pure form, without including features that lead to complex semantic interactions (like the side effects, blocks with local variables, and threads as in our IMP++).

Wadler [86] proposes a language design experiment similar in spirit to our extensions of IMP. His core language is purely functional, but some of its added features overlap with those of IMP and IMP++: state and side effects, output, non-determinism. Wadler’s objective in [86] was to emphasize the elegance and usefulness of monads in implementing interpreters in pure functional languages like Haskell, but his motivations for doing so are similar to ours: the existing approaches for programming language design are not modular enough. Monads can also be used to add modularity to denotational semantics, to avoid having to modify the mathematical domains into products of domains as we did in this section. The monadic approach to denotational semantics will be further discussed in Section 3.9. However, as also discussed in Section 3.4.2, the denotational approach to non-determinism and concurrency is to collect all possible behaviors, hereby programs evaluating to sets or lists of values. The same holds true in Wadler’s monadic approach to implement interpreters in [86]. The problem with this is that the resulting interpreters or executable semantics are quite inefficient. Contrast that with the small-step SOS approach in Section 3.3 which allows us, for example using our implementation of it in Maude, to both execute programs non-deterministically, making a possibly non-deterministic choice at each step, and search for all possible behaviors of programs.

### 3.5.9 Exercises

#### Variable Increment

The exercises below refer to the IMP extension with variable increment discussed in Section 3.5.1.

**Exercise 84.** Add variable increment to IMP, using big-step SOS:

1. Write the complete big-step SOS as a proof system;

2. Translate the proof system at 1 above into a rewrite logic theory, like in Figure 3.8;

3. ★ Implement in Maude the rewrite logic theory at 2 above, like in Figure 3.9. To test it, add to the IMP programs in Figure 3.4 the following two:

```plaintext
op sum++Pgm : -> Pgm .
eq sum++Pgm = (int m, n, s ; n = 100 ; while (++ m <= n) {
    s = s + m ;
} ) .
```

The first program should have only one behavior, so, for example, both Maude commands below should show the same result configuration, `< m |-> 101 & n |-> 100 & s |-> 5050 >`. The second program should have (only) three different behaviors under big-step semantics; the first command below will show one of the three behaviors, but the second will show all three of them:

```plaintext
rewrite < sum++Pgm > .
search < sum++Pgm > =>! Cfg:Configuration .
```

```plaintext
op nondet++Pgm : -> Pgm .
eq nondet++Pgm = (int x ; x = 1 ; x = ++ x / (++ x / x) ; ) .
```

The second command below will show one of the three behaviors, but the second will show all three of them:

```plaintext
rewrite < nondet++Pgm > .
search < nondet++Pgm > =>! Cfg:Configuration .
```
The three behaviors captured by the big-step SOS discussed in Section 3.5.1 result in configurations $< x |\rightarrow 1 >$, $< x |\rightarrow 2 >$, and $< x |\rightarrow 3 >$. As explained in Section 3.5.1, this big-step SOS should not be able to expose the behaviors in which $x$ is 0 and in which a division by zero takes place.

**Exercise 85.** Type IMP extended with variable increment:

1. Translate the big-step rule above into a rewrite logic rule that can be added to those in Figure 3.11 corresponding to the type system of IMP;

2. ★ Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the two additional programs in Example 84.

**Exercise 86.** Same as Exercise 84 but for small-step SOS instead of big-step SOS.
★ Make sure that the small-step definition in Maude exhibits all five behaviors of program nondet++Pgm defined in Exercise 84 (the three behaviors exposed by the big-step definition in Maude in Exercise 84 plus one where $x$ is 0 and one where the program gets stuck right before a division by zero).

**Exercise 87.** Same as Exercise 84 but for denotational semantics instead of big-step SOS.
★ The definition in Maude should lack any non-determinism, so only one behavior should be observed for any program, including nondet++Pgm in Exercise 84.

**Input/Output**

The exercises below refer to the IMP extension with input/output discussed in Section 3.5.2.

**Exercise 88.** Add input/output to IMP, using big-step SOS:

1. Write the complete big-step SOS as a proof system;

2. Translate the above into a rewrite logic theory, like in Figure 3.8;

3. ★ Implement the resulting rewrite logic theory in Maude, like in Figure 3.9. To test it, add to the IMP programs in Figure 3.4 the following three:

```plaintext
op sumIOPgm : -> Pgm .
  eq sumIOPgm = (int m, n, s ;
  n = read() ;
  while (m <= n) {
    print(m) ;
    s = s + m ;
    m = m + 1 ;
  }
  print(s) ; ) .

op whileIOPgm : -> Pgm .
  eq whileIOPgm = (int s ;
  s = 0 ;
  while (!(read() <= 0)) {
    s = s + read() ;
  }
  print(s) ; ) .

op nondetIOStmt : -> Stmt .
  eq nondetIOStmt = (print(read() / (read()) / read())) ;
```

The first two programs are deterministic, so both the rewrite and the search commands should only show one solution. The initial configuration in which the first program is executed should contain at least one integer in the input buffer; otherwise it does not evaluate; for example, the initial configuration $< \text{sumIOPgm}, 100 >$ yields a result configuration whose input buffer is empty and whose output buffer contains the numbers 0,1,...,100,5050. The initial configuration in which the second program is executed should eventually contain some 0 on an odd position in the input buffer; for example, $< \text{whileIOPgm}, 10 : 1 : 17 : 2 : 21 : 3 : 0 : 5 : 8 : - 2 : - 5 : 10 >$ yields a result configuration whose input buffer still contains the remaining input 5:8:-2:-5:10 and whose output buffer contains only the
integer 6. The third program, which is actually a statement, is non-deterministic. Unfortunately, big-step SOS misses behaviors because, as explained, it only achieves a non-deterministic choice semantics. For example, the commands

\[
\begin{align*}
\text{rewrite} & \quad \text{nondetIOStmt, .State, 10 : 20 : 30} \\
\text{search} & \quad \text{nondetIOStmt, .State, 10 : 20 : 30} \Rightarrow \text{Cfg:Configuration} \\
\text{yield configurations} & \quad \text{.State, epsilon, 10} \text{ and, respectively,} \quad \text{.State, epsilon, 10} \text{ and} \\
& \quad \text{.State, epsilon, 15}.
\end{align*}
\]

The configuration whose output is 6 (i.e., \(20/\ln(30/\ln 10)\)) and the three undefined configurations due to division by zero are not detected.

**Exercise 89.** Type IMP extended with input/output:

1. Translate the discussed big-step SOS typing rules for input/output into corresponding rewrite logic rules that can be added to the already existing rewrite theory in Figure 3.11 corresponding to the type system of IMP;

2. ★ Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 88.
1. Translate the big-step SOS typing rule of `halt` into a corresponding rewrite logic rule that can be added to the already existing rewrite logic theory in Figure 3.11.

2. ★ Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 92.

**Exercise 95.** Same as Exercise 92 but for the small-step SOS instead of big-step SOS.

**Exercise 96.** Same as Exercise 92 but use the top construct approach instead of the rule (SmallStep-Halting) as discussed right after the rule (SmallStep-Halting) is introduced.

**Exercise 97.** Same as Exercise 93 but eliminate the third rule of the top construct to avoid wasting any computational step.

**Exercise 98.** One could argue that the introduction of the halting configurations `<halting, σ>` was unnecessary, because we could have instead used the already existing configurations of the form `<halt, σ>`. Give an alternative small-step SOS definition of abrupt termination which does not add special halting configurations. Can we avoid the introduction of the top construct discussed above? Comment on the disadvantages of this approach.

**Exercise 99.** Same as Exercise 92 but for denotational semantics instead of big-step SOS.

**Exercise 100.** Same as Exercise 99 but modifying the denotation of assignment so that it is always undefined when the assigned variable has not been declared.

**Dynamic Threads**

The exercises below refer to the IMP extension with dynamic threads discussed in Section 3.5.4.

**Exercise 101.** Consider the two programs at the beginning of Section 3.5.4. Propose hypothetical executions of the second program corresponding to any of the seven possible values of `x`. What is the maximum value that `s` can have when the first program, as well as all its dynamically created threads, terminate? Is there some execution of the first program corresponding to each smaller value of `s` (but larger than or equal to 0)?

**Exercise 102.** Add dynamic threads to IMP, using big-step SOS:

1. Write the complete big-step SOS as a proof system;

2. Translate the above into a rewrite logic theory, like in Figure 3.8;

3. ★ Implement the resulting rewrite logic theory in Maude, like in Figure 3.9. To test it, execute the two programs at the beginning of Section 3.5.4. The resulting Maude big-step SOS definition may be slow on the first program for large initial values for `n`: even though it does not capture all possible behaviors, it still comprises many of them. For example, searching for all the result configurations when `n = 10` gives 12 possible values for `s`, namely 55, 56, ..., 66. On the other hand, the second program only shows one out of the seven behaviors, namely the one where `x` results in 111.

**Exercise 103.** Same as Exercise 102 but for the type system instead of big-step SOS.
Exercise 104. Same as Exercise 102, but for small-step SOS instead of big-step SOS. When representing the resulting small-step SOS into rewrite logic, the structural identity can be expressed as a rewrite logic equation, this way capturing faithfully the intended computational granularity of the small-step SOS. ★ When implementing the resulting rewrite logic theory into Maude, this equation can either be added as a normal equation (using \texttt{eq}) or as an \texttt{assoc} attribute to the sequential composition construct. The former will only be applied from left-to-right when executed using Maude rewriting and search commands, but that is sufficient in our particular case here.

Exercise 105. Same as Exercise 102, but for denotational semantics instead of big-step SOS.

Local Variables

The exercises below refer to the IMP extension with blocks and local variables discussed in Section 3.5.5.

Exercise 106. ★ Implement in Maude the seven macros in the preamble of Section 3.5.5, which desugar blocks with local variable declarations into \texttt{let} constructs. 

Hint: In Maude, equations are applied in order. One should typically not rely on that, but in this case it may give us a simpler and more compact implementation.

Exercise 107. Add blocks with local variables to IMP, using big-step SOS:

1. Write the complete big-step SOS as a proof system;

2. Translate the above into a rewrite logic theory, like in Figure 3.8;

3. ★ Implement the resulting rewrite logic theory in Maude, like in Figure 3.9. To test it, modify the IMP programs in Figure 3.4 to use local variables and also add the two programs at the beginning of Section 3.5.5. To check whether the programs evaluate as expected, you can let some relevant variables purposely undeclared and bind them manually (to 0) in the initial state. When the programs terminate, you will see the new values of those variables. If you execute closed programs (i.e., programs declaring all the variables they use) then the resulting states will be empty, because our semantics of \texttt{let} recovers the state, so it will be difficult or impossible to know whether they executed correctly.

Exercise 108. Same as Exercise 107, but for the type system instead of big-step SOS.

Exercise 109. Same as Exercise 107, but for small-step SOS instead of big-step SOS.

Exercise 110. Same as Exercise 107, but for denotational semantics instead of big-step SOS.

Putting Them All Together

The exercises below refer to the IMP extension with blocks and local variables discussed in Section 3.5.6.

Exercise 111. Define IMP++ using big-step SOS, assuming that abrupt termination applies to the entire program, no matter whether the abrupt termination signal has been issued from inside a spawned thread or from the main program, and assuming that nothing special is done to enhance the parallelism of \texttt{spawn} within \texttt{let} blocks:

1. Write the complete big-step SOS as a proof system;

2. Translate the above into a rewrite logic theory, like in Figure 3.8;
3. ★ Implement the resulting rewrite logic theory in Maude, like in Figure 3.9. To test it, propose five tricky IMP++ programs. You programs will be added to our test-suite before grading this exercise. You get extra-points if your programs reveal limitations in the number of behaviors captured by other students’ definitions.

**Exercise 112.** Same as Exercise 111 but assume that abrupt termination applies to the issuing thread only, letting the rest of the threads continue. The main program is considered to be a thread itself.

**Exercise 113.** Same as Exercise 111 but for the type system instead of big-step SOS.

**Exercise 114.** Same as Exercise 111 but for small-step SOS instead of big-step SOS.

**Exercise 115.** Same as Exercise 112 but for small-step SOS instead of big-step SOS.

**Exercise* 116.** Same as Exercise 114 but define and use “parallel configurations” of the form C∥ in order to enhance the parallelism of spawn statements from inside let blocks.

**Exercise* 117.** Same as Exercise 116 but eliminate the let construct when semantically unnecessary using a substitution operation (which needs to also be defined).

**Exercise* 118.** Same as Exercise 117 but use an environment-store approach to the state instead of substitution.

**Exercise 119.** Same as Exercise 111 but for denotational semantics instead of big-step SOS.
Anything else to add here?

Mention Moggi/Wadler among those who criticized plain SOS and plain, non-monadic denotational semantics?

Give a bit of history on who split for the first time a state into an environment and a store, as an alternative to using substitutions.
VERY IMPORTANT!

Make IMP have the following syntax:

\[
\begin{align*}
\text{Int} &::= \text{the domain of (unbounded) integer numbers, with usual operations on them} \\
\text{Bool} &::= \text{the domain of Booleans} \\
\text{Id} &::= \text{standard identifiers} \\
\text{AExp} &::= \text{Int} \\
&\quad | \quad \text{Id} \\
&\quad | \quad \text{AExp} + \text{AExp} \\
&\quad | \quad \text{AExp} / \text{AExp} \\
\text{BExp} &::= \text{Bool} \\
&\quad | \quad \text{AExp} \leq \text{AExp} \\
&\quad | \quad \text{not BExp} \\
&\quad | \quad \text{BExp} \text{ and BExp} \\
\text{Stmt} &::= \{\} \\
&\quad | \quad \{\text{Stmt}\} \\
&\quad | \quad \text{Id} = \text{AExp}; \\
&\quad | \quad \text{Stmt} \text{ Stmt} \\
&\quad | \quad \text{if(\text{BExp}) Stmt} \text{ else Stmt} \\
&\quad | \quad \text{while(\text{BExp})Stmt} \\
\text{Pgm} &::= \text{int List\{Id\};Stmt}
\end{align*}
\]

Add the following to IMP++, removing the above production for Pgm:

\[
\begin{align*}
\text{AExp} &::= \text{++Id} \\
&\quad | \quad \text{read()} \\
&\quad | \quad \text{spawnStmt} \\
\text{Stmt} &::= \text{AExp;} \\
&\quad | \quad \text{print(AExp);} \\
&\quad | \quad \text{halt;} \\
&\quad | \quad \text{join(AExp)} \\
&\quad | \quad \text{int List\{Id\};} \\
\text{Pgm} &::= \text{Stmt}
\end{align*}
\]

To make this language design experiment more interesting, first consider spawn to be a statement construct, as it is right now all over this chapter. This way we also keep all this material.

Then argue that we would like to also be able to synchronize somehow with the spawned thread, at least to join it. In this case, we need an id or something similar to be returned by the spawn command. Then propose to make spawn an expression construct as above, which returns an integer. Then comment what it takes for each style to support this change. Finally, make all Maude code to consider this as the final IMP++ language.
Modular structural operational semantics (MSOS) was introduced, as its name implies, to address the non-modularity aspects of (big-step and/or small-step) SOS. As already seen in Section 3.5, there are several reasons why big-step and small-step SOS are non-modular, as well as several facets of non-modularity in general. In short, a definitional framework is non-modular when, in order to add a new feature to an existing language or calculus, one needs to revisit and change some or all of the already defined unrelated features. For example, recall the IMP extension with input/output in Section 3.5.2. We had to add new semantic components in the IMP configurations, both in the big-step and in the small-step SOS definitions, to hold the input/output buffers. That meant, in particular, that all the existing big-step and/or small-step SOS rules of IMP had to change. That was, at best, very inconvenient.

Before we get into the technicalities of MSOS, one natural question to address is why we need modularity of language definitions. One may argue that defining a programming language is a major endeavor, done once and for all, so having to go through the semantic rules many times is, after all, not such a bad idea, because it gives one the chance to find and fix potential errors in them. Here are several reasons why modularity is desirable in language definitions, in no particular order:

- Having to modify many or all rules whenever a new rule is added that modifies the structure of the configuration is actually more error prone than it may seem, because rules become heavier to read and debug; for example, one can write $\sigma$ instead of $\sigma'$ in a right-hand-side of a rule and a different or wrong language is defined.

- A modular semantic framework allows us to more easily reuse semantics of existing and probably already well-tested features in other languages or language extensions, thus increasing our productivity as language designers and our confidence in the correctness of the resulting language definition.

- When designing a new language, as opposed to an existing language, one needs to experiment with features and combinations of features; having to do unrelated changes whenever a new feature is added to or removed from the language burdens the language designer with boring tasks taking considerable time that could have been otherwise spent on actual interesting language design issues.

- There is an increasing number of domain-specific languages, resulting from the need to abstract away from low-level programming language details to important, domain-specific application aspects. Hence, there is a need for language design and experimentation for various domains. Moreover, domain-specific languages tend to be dynamic, being added or removed features frequently as the domain knowledge evolves. It would be nice to have the possibility to “drag-and-drop” features in one’s language, such as functions, exceptions, etc.; however, modularity is crucial for that.

To our knowledge, MSOS was the first framework that explicitly recognizes the importance of modular language design and provides explicit support to achieve it in the context of SOS. Reduction semantics with evaluation contexts (see Section 3.7) was actually proposed before MSOS and also offers modularity in language semantic definitions, but its modularity comes as a consequence of a different way to propagate reductions through language constructs and not as an explicit goal that it strives to achieve.

There are both big-step and small-step variants of MSOS, but we discuss only small-step MSOS here. We actually generically call MSOS the small-step, implicitly-modular variant of MSOS (see Section 3.6.4). To bring modularity to SOS, MSOS proposes the following:

- Separate the syntax (i.e., the fragment of program under consideration) from the non-syntactic components in configurations, and treat them differently, as explained below;
• Make the transitions only relate syntax to syntax (as opposed to configurations), and hide the non-
syntactic components in a transition label, as explained below;
• Encode in the transition label all the changes in the non-syntactic components of the configuration that
need to be applied together with the syntactic reduction given by the transition;
• Use specialized notation in transition labels together with a discipline to refer to the various semantic
components and to say that some of them stay unchanged; also, labels can be explicitly or implicitly
shared by the conditions and the conclusion of a rule, elegantly capturing the idea that “changes are
propagated” through desired language constructs.

A transition in MSOS is of the form

\[ P \xrightarrow{\Delta} P' \]

where \( P \) and \( P' \) are programs or fragments of programs and \( \Delta \) is a label describing the semantic configuration
components both before and after the transition. Specifically, \( \Delta \) is a record containing fields denoting the
semantic components of the configuration. The preferred notation in MSOS for stating that in label \( \Delta \) the
semantic component associated to the field name field before the transition takes place is \( \alpha \) is
\( \Delta = \{ \text{field} = \alpha, \ldots \} \). Similarly, the preferred notation for stating that the semantic component associated to field field after
the transition takes place is \( \beta \) is
\( \Delta = \{ \text{field'} = \beta, \ldots \} \) (the field name is primed). For example, the second
MSOS rule for variable assignment (when the assigned arithmetic expression is already evaluated) is (this is
rule (MSOS-Asgn) in Figure 3.24):

\[ x = i; \{ \text{state} = \sigma, \text{state'} = \sigma[i/x], \ldots \} \rightarrow \{ \} \text{ if } \sigma(x) \neq \bot \]

It is easy to desugar the rule above into a more familiar SOS rule of the form:

\[ \langle x = i; \sigma \rangle \rightarrow \langle \{ \}, \sigma[i/x] \rangle \text{ if } \sigma(x) \neq \bot \]

The above is precisely the rule (SmallStep-Asgn) in the small-step SOS of IMP (see Figure 3.15). The MSOS
rule is actually more modular than the SOS one, because of the “…”, which means that everything else in the
configuration stays unchanged. For example, if we want to extend the language with input/output language
constructs as we did in Section 3.5.2, then new semantic components, namely the input and output buffers, need to be added to the configuration. Moreover, as seen in Section 3.5.2 the SOS rule above needs to be changed into a rule of the form

\[ \langle x = i; \sigma, \omega_{in}, \omega_{out} \rangle \rightarrow \langle \{ \}, \sigma[i/x], \omega_{in}, \omega_{out} \rangle \text{ if } \sigma(x) \neq \bot \]

where \( \omega_{in} \) and \( \omega_{out} \) are the input and output buffers, respectively, which stay unchanged during the variable
assignment operation, while the MSOS rule does not need to be touched.

To impose a better discipline on the use of labels, at the same time making the notation even more
compact, MSOS splits the fields into three categories: read-only, read-write, and write-only. The field state
above was read-write, meaning that the transition label can both read its value before the transition takes
place and write its value after the transition takes place. Unlike the state, which needs to be both read and
written, there are semantic configuration components that only need to be read, as well as ones that only need
to be written. In these cases, it is recommended to use read-only or write-only fields.

Read-only fields are only inspected by the rule, but not modified, so they only appear unprimed in labels.
For example, the following can be one of the MSOS rules for the let binding language construct in a pure
Indeed, transitions do not modify the environment in a pure functional language. They only use it in a read-only fashion to lookup the variable values. A new environment is created in the premise of the rule above to reduce the body of the let, but neither of the transitions in the premise nor in the conclusion of the rule change their environment. Note that this was not the case for IMP extended with let in Section 3.5.5 because there we wanted blocks and local variable declarations to desugar to let statements. Since we allowed variables to be assigned new values in blocks, like in conventional imperative and object-oriented languages, we needed an impure variant of let. As seen in Section 3.6.2, our MSOS definition of IMP++’s let uses a read-write attribute (the state attribute). We do not discuss read-only fields any further here.

Write-only fields are used to record data that is not analyzable during program execution, such as the output or the trace. Their names are always primed and they have a free monoid semantics—everything written on them is actually added to the end (see [53] for technical details). Consider, for example, the extension of IMP with an output (or print) statement in Section 3.5.2, whose MSOS second rule (after the argument is evaluated, namely rule (MSOS-Print) in Section 3.6.2) is:

\[
\text{print}(i); \quad \{ \text{output}'=i, \ldots \} \to {} \]

Compare the rule above with the one below, which uses a read-write attribute instead:

\[
\text{print}(i); \quad \{ \text{output}=?\text{out}, \text{output}'=\text{out} : i, \ldots \} \to {} \]

Indeed, mentioning the \(\omega_{\text{out}}\) like in the second rule above is unnecessary, error-prone (e.g., one may forget to add it to the primed field or may write \(i : \omega_{\text{out}}\) instead of \(\omega_{\text{out}} : i\)), and non-modular (e.g., one may want to change the monoid construct, say to write \(\omega_{\text{out}} \cdot i\) instead of \(\omega_{\text{out}} : i\), etc.).

MSOS achieves modularity in two ways:

1. By making intensive use of the record comprehension notation “…”, which, as discussed, indicates that more fields could follow but that they are not of interest. In particular, if the MSOS rule has no premises, like in the rules for the assignment and print statements discussed above, than the “…” says that the remaining contents of the label stays unchanged after the application of the transition; and

2. By reusing the same label or portion of label both in the premise and in the conclusion of an MSOS proof system rule. In particular, if “…” is used in the labels of both the premise and the conclusion of an MSOS rule, then all the occurrences of “…” stand for the same portion of label, that is, the same fields bound to the same semantic components.

For example, the following MSOS rules for first-statement reduction in sequential composition are equivalent
and say that all the changes generated by reducing $s_1$ to $s'_1$ are propagated when reducing $s_1 s_2$ to $s'_1 s_2$:

$$
\begin{align*}
\frac{s_1 \xrightarrow{\Lambda} s'_1}{s_1 s_2 \xrightarrow{\Lambda} s'_1 s_2} \\
\frac{s_1 \xrightarrow{\ldots} s'_1}{s_1 s_2 \xrightarrow{\ldots} s'_1 s_2} \\
\frac{s_1 \xrightarrow{\text{[state} = \sigma, \ldots\text{]}} s'_1}{s_1 s_2 \xrightarrow{\text{[state} = \sigma, \ldots\text{]}} s'_1 s_2}
\end{align*}
$$

Indeed, advancing the first statement in a sequential composition of statements one step has the same effect on the configuration as if the statement was advanced the same one step in isolation, without the other statement involved; said differently, the side effects are all properly propagated.

MSOS (the implicitly-modular variant of it, see Section 3.6.4) has been refined to actually allow for dropping such redundant labels like above from rules. In other words, if a label is missing from a transition then the implicit label is assumed: if the rule is unconditional then the implicit label is the identity label (in which the primed fields have the same values as the corresponding unprimed ones, etc.), but if the rule is conditional then the premise and the conclusion transitions share the same label, that is, they perform the same changes on the semantic components of the configuration. With this new notational convention, the most elegant and compact way to write the rule above in MSOS is:

$$
\frac{s_1 \xrightarrow{\Lambda} s'_1}{s_1 s_2 \xrightarrow{} s'_1 s_2}
$$

This is precisely the rule (MSOS-SEQ-ARG1) in Figure 3.24, part of the MSOS semantics of IMP.

One of the important merits of MSOS is that it captures formally many of the tricks that language designers informally use to avoid writing awkward and heavy SOS definitions.

Additional notational shortcuts are welcome in MSOS if properly explained and made locally rigorous, without having to rely on other rules. For example, the author of MSOS finds the rule

$$
\frac{x \xrightarrow{\text{[state}, \ldots\text{]}} \text{state}(x) \text{ if state}(x) \neq \perp}{x \xrightarrow{} \sigma(x) \text{ if } \sigma(x) \neq \perp}
$$

deeply. However, the fact that, strictly speaking, \text{state}(x) does not make sense by itself (recall that state is a field name, not the state) and that field names are expected to be paired with their semantic components in labels. Nevertheless, there is only one way to make sense of this rule, namely to replace any use of state by its semantic contents, which therefore does not need to be mentioned.

A major goal when using MSOS to define languages or calculi is to write on the labels as little information as possible and to use the implicit conventions for the missing information. That is because everything written on labels may work against modularity if the language is later on extended or simplified. As an extreme case,
if one uses only read/write fields in labels and mentions all the fields together with all their semantic contents on every single label, then MSOS becomes conventional SOS and therefore suffers from the same limitations as SOS with regards to modularity.

Recall the rules in Figure [3.16] for deriving the transitive closure \( \rightarrow^* \) of the small-step SOS relation \( \rightarrow \). In order for two consecutive transitions to compose, the source configuration of the second had to be identical to the target configuration of the first. A similar property must also hold in MSOS, otherwise one may derive inconsistent computations. This process is explained in MSOS by making use of category theory (see [53] for technical details on MSOS; see Section 2.7 for details on category theory), associating MSOS labels with morphisms in a special category and then using the morphism composition mechanism of category theory.

However, category theory is not needed in order to understand how MSOS works in practice. A simple way to explain its label composition is by translating, or desugaring MSOS definitions into SOS, as we implicitly suggested when we discussed the MSOS rule for variable assignment above. Indeed, once one knows all the fields in the labels, which happens once a language definition is complete, one can automatically associate a standard small-step SOS definition to the MSOS one by replacing each MSOS rule with an SOS rule over configurations including, besides the syntactic contents, the complete semantic contents extracted from the notational conventions in the label. The resulting SOS configurations will not have fields anymore, but will nevertheless contain all the semantic information encoded by them. For example, in the context of a language containing only a state and an output buffer as semantic components in its configuration (note that IMP++ contained an input buffer as well), the four rules discussed above for variable assignment, output, sequential composition, and lookup desugar, respectively, into the following conventional SOS rules:

\[
\begin{align*}
\langle x = i; \sigma, \omega \rangle & \rightarrow \langle \{\}, \sigma[i/x], \omega \rangle \quad \text{if } \sigma(x) \neq \bot \\
\langle \text{print}(i); \sigma, \omega \rangle & \rightarrow \langle \{\}, \sigma, \omega : i \rangle \\
\langle s_1, \sigma, \omega \rangle & \rightarrow \langle s_1', \sigma', \omega' \rangle \\
\langle s_1 s_2, \sigma, \omega \rangle & \rightarrow \langle s_1' s_2', \sigma', \omega' \rangle \\
\langle x, \sigma, \omega \rangle & \rightarrow \langle \sigma(x), \sigma, \omega \rangle \quad \text{if } \sigma(x) \neq \bot
\end{align*}
\]

Recall that for unconditional MSOS rules the meaning of the missing label fields is “stay unchanged”, while in the case of conditional rules the meaning of the missing fields is “same changes in conclusion as in the premise”. In order for all the changes explicitly or implicitly specified by MSOS rules to apply, one also needs to provide an initial state for all the attributes, or in terms of SOS, an initial configuration. The initial configuration is often left unspecified in MSOS or SOS paper language definitions, but it needs to be explicitly given when one is concerned with executing the semantics. In our SOS definition of IMP in Section 3.3.2 (see Figure 3.15), we created the appropriate initial configuration in which the top-level statement was executed using the proof system itself, more precisely the rule (SmallStep-Pgm) created a configuration holding a statement and a state from a configuration holding only the program. That is not possible in MSOS, because MSOS assumes that the structure of the label record does not change dynamically as the rules are applied. Instead, it assumes all the attributes given and fixed. Therefore, one has to explicitly state the initial values corresponding to each attribute in the initial state. However, in practice those initial values are understood and, consequently, we do not bother defining them. For example, if an attribute holds a list or a set, then its initial value is the empty list or set; if it holds a partial function, then its initial value is the partial function undefined everywhere; etc.

This way of regarding MSOS as a convenient front-end to SOS also supports the introduction of further
notational conventions in MSOS if desired, like the one discussed above using \( \text{state}(x) \) instead of \( \sigma(x) \) in the right-hand-side of the transition, provided that one explains how such conventions are desugared when going from MSOS to SOS. Finally, the translation of MSOS into SOS also allows MSOS to borrow from SOS the reflexive/transitive closure \( \rightarrow^* \) of the one-step relation.

3.6.1 The MSOS of IMP

Figures 3.23 and 3.24 show the MSOS definition of IMP. There is not much to comment on the MSOS rules in these figures, except, perhaps, to note how compact and elegant they are compared to the corresponding SOS definition in Figures 3.14 and 3.15. Except for the three rules (MSOS-Lookup), (MSOS-Asgn), and (MSOS-Var), which make use of labels, they are as compact as they can be in any SOS-like setting for any language including the defined constructs. Also, the above-mentioned three rules only mention those components from the labels that they really need, so they allow for possible extensions of the language, like the IMP++ extension in Section 3.5.

The rule (MSOS-Var) is somehow different from the other rules that need the information in the label, in that it uses an attribute which has the type read-write but it only writes it without reading it. This is indeed possible in MSOS. The type of an attribute cannot be necessarily inferred from the way it is used in some of the rules, and not all rules must use the same attribute in the same way. One should explicitly clarify the type of each attribute before one gives the actual MSOS rules, and one is not allowed to change the attribute types dynamically, during derivations. Indeed, if the type of the output attribute in the MSOS rules for output above (and also in Section 3.6.2) were read-write, then the rules would wrongly imply that the output buffer will only store the last value, the previous ones being lost (this could be a desirable semantics in some cases).

Since the MSOS proof system in Figures 3.23 and 3.24 translates, following the informal procedure described above, in the SOS proof system in Figures 3.14 and 3.15, basically all the small-step SOS intuitions and discussions for IMP in Section 3.3.2 carry over here almost unchanged. In particular:

**Definition 22.** We say that \( C \rightarrow C' \) is derivable with the MSOS proof system in Figures 3.23 and 3.24 written \( \text{MSOS}(\text{IMP}) \vdash C \rightarrow C' \) if \( \text{SmallStep}(\text{IMP}) \vdash C \rightarrow C' \) (using the proof system in Figures 3.14 and 3.15). Similarly, \( \text{MSOS}(\text{IMP}) \vdash C \rightarrow^* C' \) if \( \text{SmallStep}(\text{IMP}) \vdash C \rightarrow^* C' \).

Note, however, that MSOS is more syntactic in nature than SOS, in that each of its reduction rules requires syntactic terms in both sides of the transition relation. In particular, that means that, unlike in SOS (see Exercise 68), in MSOS one does not have the option to dissolve statements from configurations anymore. Instead, one needs to reduce them to \( \{ \} \) or some similar syntactic constant; if the original language did not have such a constant then one needs to invent one and add it to the original language or calculus syntax.

3.6.2 The MSOS of IMP++

We next discuss the MSOS of IMP++, playing the same language design scenario as in Section 3.5: we first add each feature separately to IMP, as if that feature was the final extension of the language, and then we add all the features together and investigate the modularity of the resulting definition as well as possibly unexpected feature interactions.

**Variable Increment**

In MSOS, one can define the increment modularly:

\[
\begin{align*}
\text{++ } x & \quad \xrightarrow{\text{state}=r, \ \text{state}'=\sigma\{\sigma(x)+\text{int}1\}/x} \\
& \rightarrow \sigma(x) + \text{int}1 \quad (\text{MSOS-Inc})
\end{align*}
\]
\[
\begin{align*}
\text{if } \sigma(x) \not= \bot & \quad \text{(MSOS-LOOKUP)} \\
\frac{a_1 \to a'_1}{a_1 + a_2 \to a'_1 + a_2} & \quad \text{(MSOS-ADD-ARG1)} \\
\frac{a_2 \to a'_2}{a_1 + a_2 \to a'_1 + a'_2} & \quad \text{(MSOS-ADD-ARG2)} \\
i_1 + i_2 & \to i_1 +_{\text{int}} i_2 \quad \text{(MSOS-ADD)} \\
\frac{a_1 \to a'_1}{a_1 / a_2 \to a'_1 / a_2} & \quad \text{(MSOS-DIV-ARG1)} \\
\frac{a_2 \to a'_2}{a_1 / a_2 \to a'_1 / a'_2} & \quad \text{(MSOS-DIV-ARG2)} \\
i_1 / i_2 & \to i_1 /_{\text{int}} i_2 \quad \text{if } i_2 \not= 0 \quad \text{(MSOS-DIV)} \\
\frac{a_1 \to a'_1}{a_1 \leq a_2 \to a'_1 \leq a_2} & \quad \text{(MSOS-LEQ-ARG1)} \\
\frac{a_2 \to a'_2}{i_1 \leq a_2 \to i_1 \leq a'_2} & \quad \text{(MSOS-LEQ-ARG2)} \\
i_1 \leq i_2 & \to i_1 \leq_{\text{int}} i_2 \quad \text{(MSOS-LEQ)} \\
\frac{b \to b'}{! b \to ! b'} & \quad \text{(MSOS-NOT-ARG)} \\
\text{! true} & \to \text{false} \quad \text{(MSOS-NOT-TRUE)} \\
\text{! false} & \to \text{true} \quad \text{(MSOS-NOT-FALSE)} \\
\frac{b_1 \to b'_1}{b_1 \& b_2 \to b'_1 \& b_2} & \quad \text{(MSOS-AND-ARG1)} \\
\text{false} \& b_2 & \to \text{false} \quad \text{(MSOS-AND-FALSE)} \\
\text{true} \& b_2 & \to b_2 \quad \text{(MSOS-AND-TRUE)}
\end{align*}
\]

Figure 3.23: MSOS(IMP) — MSOS of IMP Expressions \((i_1, i_2 \in \text{Int}; x \in \text{Id}; a_1, a'_1, a_2, a'_2 \in \text{AExp}; b, b', b_1, b'_1, b_2 \in \text{BExp}; \sigma \in \text{State})\).
\[
\begin{align*}
& a \rightarrow a' \\
& x = a; \rightarrow x = a'; \quad \text{(MSOS-ASGN-ARG2)} \\
& x = i; \xrightarrow{[\text{state}=\sigma, \text{state}'=\sigma[i/x], \ldots]} \{ \} \quad \text{if } \sigma(x) \neq \bot \quad \text{(MSOS-ASGN)} \\
& s_1 \rightarrow s'_1 \quad \text{(MSOS-SEQ-ARG1)} \\
& s_1 \rightarrow s'_1 \quad \text{(MSOS-SEQ-ARG2)} \\
& \{ \} s_2 \rightarrow s_2 \quad \text{(MSOS-SEQ-Skip)} \\
& b \rightarrow b' \\
& \text{if } (b) s_1 \text{ else } s_2 \rightarrow \text{if } (b') s_1 \text{ else } s_2 \quad \text{(MSOS-IF-ARG1)} \\
& \text{if } (\text{true}) s_1 \text{ else } s_2 \rightarrow s_1 \quad \text{(MSOS-IF-TRUE)} \\
& \text{if } (\text{false}) s_1 \text{ else } s_2 \rightarrow s_2 \quad \text{(MSOS-IF-FALSE)} \\
& \text{while } (b) s \rightarrow \text{if } (b) \{ s \text{ while } (b) s \} \{ \} \quad \text{(MSOS-WHILE)} \\
& \text{int } xl; s \xrightarrow{[\text{state}'=xl \rightarrow 0, \ldots]} s \quad \text{(MSOS-VAR)}
\end{align*}
\]

Figure 3.24: MSOS(IMP) — MSOS of IMP Statements (i \in \text{Int}; x \in \text{Id}; xl \in \text{List(Id)}; a, a' \in \text{AExp}; b, b' \in \text{BExp}; s, s_1, s_2, s_1' \in \text{Stmt}; \sigma \in \text{State}).

No other rule needs to be changed, because MSOS already assumes that, unless otherwise specified, each rule propagates all the configuration changes in its premise(s).

**Input/Output**

MSOS can modularly support the input/output extension of IMP. We need to add new label attributes holding the input and the output buffers, say input and output, respectively, and then to add the corresponding rules for the input/output constructs. Note that the input attribute is read-write, while the output attribute is write-only. Here are the MSOS rules for input/output:

\[
\begin{align*}
& \text{read()} \xrightarrow{[\text{input}=i, \text{input}'=\omega, \ldots]} i \quad \text{(MSOS-READ)} \\
& a \rightarrow a' \\
& \text{print}(a); \rightarrow \text{print}(a') \quad \text{(MSOS-PRINT-ARG)} \\
& \text{print}(i); \xrightarrow{[\text{output}=i, \ldots]} \{ \} \quad \text{(MSOS-PRINT)}
\end{align*}
\]

Note that, since output is a write-only attribute, we only need to mention the new value that is added to the output in the label of the second rule above. If output was declared as a read-write attribute, then the label of the second rule above would have been \{output' = \omega, output = \omega : i, \ldots\}. A major implicit objective of MSOS is to minimize the amount of information that the user needs to write in each rule. Indeed, anything written by a user can lead to non-modularity and thus work against the user when changes are performed to
the language. For example, if for some reason one declared output as a read-write attribute and then later on one decided to change the list construct for the output integer list from colon “;” to something else, say “·”, then one would need to change the label in the second rule above from \{output = ω, output' = ω : i, ...\} to \{output = ω, output' = (ω · i), ...\}. Therefore, in the spirit of enhanced modularity and clarity, the language designer using MSOS is strongly encouraged to use write-only (or read-only) attributes instead of read-write attributes whenever possible.

Notice the lack of expected duality between the rules (MSOS-READ) and (MSOS-PRINT) for input and for output above. Indeed, for all the reasons mentioned above, one would like to write the rule (MSOS-READ) more compactly and modularly as follows:

\[
\text{read()} \rightarrow i
\]

Unfortunately, this is not possible with the current set of label attributes provided by MSOS. However, there is no reason why MSOS could not be extended to include more attributes. For example, an attribute called “consumable” which would behave as the dual of write-only, i.e., it would only have an unprimed variant in the label holding a monoid (or maybe a group?) structure like the read-only attributes but it would consume from it whatever is matched by the rule label, would certainly be very useful in our case here. If such an attribute type were available, then our input attribute would be of that type and our MSOS rule for read() would be like the one above.

A technical question regarding the execution of the resulting MSOS definition is how to provide input to programs. Or, put differently, how to initialize configurations. One possibility is to assume that the user is fully responsible for providing the initial attribute values. This is, however, rather inconvenient, because the user would then always have to provide an empty state and an empty output buffer in addition to the desired input buffer in each configuration. A more convenient approach is to invent a special syntax allowing the user to provide precisely a program and an input to it, and then to automatically initialize all the attributes with their expected values. Let us pair a program \( p \) and an input \( ω \) for it using a configuration-like notation of the form \( ⟨p, ω⟩ \). Then we can replace the rule (MSOS-Var) in Figure 3.24 with the following rule:

\[
⟨\text{int } x l; \ s, \ ω⟩ \rightarrow s
\]

_Abrupt Termination_

MSOS allows for a more modular semantics of abrupt termination than the more conventional semantic approaches discussed in Section 3.5.3. However, in order to achieve modularity, we need to extend the syntax of IMP with a \texttt{top} construct, similarly to the small-step SOS variant discussed in Section 3.5.3. The key to modularity here is to use the labeling mechanism of MSOS to carry the information that a configuration is in a halting status. Let us assume an additional write-only field in the MSOS labels, called halting, which is \texttt{true} whenever the program needs to halt, otherwise it is \texttt{false}7 Then we can add the following two MSOS rules that set the halting field to \texttt{true}:

\[
i_1 / 0 \rightarrow i_1 / 0 \quad \text{(MSOS-DIV-BY-ZERO)}
\]

\[
halt; \rightarrow halt; \quad \text{(MSOS-HALT)}
\]

7Strictly speaking, MSOS requires that the write-only attributes take values from a free monoid; if one wants to be faithful to that MSOS requirement, then one can replace \texttt{true} with some letter word and \texttt{false} with the empty word.
As desired, it is indeed the case now that a sequent of the form $s \vdash \{\text{halting} = \text{true}, \ldots\} \rightarrow s'$ is derivable if and only if $s = s'$ and the next executable step in $s$ is either a "halt;" statement or a division-by-zero expression. If one does not like keeping the syntax unchanged when an abrupt termination takes place, then one can add a new syntactic construct, say stuck like in [53], and replace the right-hand-side configurations above with stuck; that does not conceptually change anything in what follows. The setting seems therefore perfect for adding a rule of the form

$$
\frac{\{\text{halting} = \text{true}, \ldots\} \rightarrow s}{s \vdash s' \rightarrow \{\}}
$$

and declare ourselves done, because now an abruptly terminated statement terminates just like any other statement, with a {} statement as result and with a label containing a non-halting status. Unfortunately, that does not work, because such a rule would interfere with other rules taking statement reductions as preconditions, for example with the first precondition of the (MSOS-Seq) rule, and thus hide the actual halting status of the precondition. To properly capture the halting status, we define a top level statement construct like we discussed in the context of big-step and small-step SOS above, say top Stmt, modify the rule (MSOS-Var) from

$$\text{int } xl; \ s \vdash \{\text{state} = xl \mapsto 0, \ldots\} \rightarrow s$$

to

$$\text{int } xl; \ s \vdash \{\text{state} = xl \mapsto 0, \ \text{halting} = \text{false}, \ldots\} \rightarrow \text{top } s$$

to mark the top level statement, and then finally include the following three rules:

$$\text{top } s \vdash \{\text{halting} = \text{false}, \ldots\} \rightarrow \text{top } s' \quad \text{(MSOS-Top-Normal)}$$

$$\text{top } s \vdash \{\} \rightarrow \{\} \quad \text{(MSOS-Top-Skip)}$$

$$\text{top } s \vdash \{\text{halting} = \text{true}, \ldots\} \rightarrow s' \quad \text{top } s \vdash \{\text{halting} = \text{false}, \ldots\} \rightarrow \{\} \quad \text{(MSOS-Top-Halting)}$$

The use of a top construct like above seems unavoidable if we want to achieve modularity. Indeed, we managed to avoid it in the small-step SOS definition of abrupt termination in Section §3.5.3 (paying one additional small-step to dissolve the halting configuration), because the halting configurations were explicitly, and thus non-modularly propagated through each of the language constructs, so the entire program reduced to a halting configuration whenever a division by zero or a "halt;" statement was encountered. Unfortunately, that same approach does not work with MSOS (unless we want to break its modularity, like in SOS), because the syntax is not mutilated when an abrupt termination occurs. The halting signal is captured by the label of the transition. However, the label does not tell us when we are at the top level in order to dissolve the halting status. Adding a new label to hold the depth of the derivation, or at least whether we are the top or not, would require one to (non-modularly) change it in each rule. The use of an additional top construct like we did above appears to be the best trade-off between modularity and elegance here.

Note that, although MSOS can be mechanically translated into SOS by associating to each MSOS attribute an SOS configuration component, the solution above to support abrupt termination modularly in MSOS is not
modular when applied in SOS via the translation. Indeed, adding a new attribute in the label means adding
a new configuration component, which already breaks the modularity of SOS. In other words, the MSOS
technique above cannot be manually used in SOS to obtain a modular definition of abrupt termination in SOS.

Dynamic Threads

The small-step SOS rules for spawning threads in Section 3.5.4 straightforwardly turn into MSOS rules:

\[
s \rightarrow s' \\
\text{spawn } s \rightarrow \text{spawn } s' \\
\text{spawn } \{ \} \rightarrow \{ \} \\
\text{(spawn } s_1) s_2 \rightarrow (\text{spawn } s_1) s_2' \\
(s_1 s_2) s_3 \equiv s_1 (s_2 s_3) \\
\]

(MSOS-\text{SPAWN-ARG})
(MSOS-\text{SPAWN-SKIP})
(MSOS-\text{SPAWN-WAIT})
(MSOS-\text{SEQ-ASSOC})

\text{Stmt ::= | \text{spawn Stmt}}

Even though the MSOS rules above are conceptually identical to the original small-step SOS rules, they are
more modular because, unlike the former, they carry over unchanged when the configuration needs to change.
Note that the structural identity stating the associativity of sequential composition, called (MSOS-\text{SEQ-ASSOC})
avbove, is still necessary, and so is the syntactic extension of \text{spawn} to take statements instead of blocks.

Local Variables

Section 3.5.5 showed how blocks with local variables can be desugared into a uniform \text{let} construct, and
also gave the small-step SOS rules defining the semantics of \text{let}. Those rules can be immediately adapted
into the following MSOS rules:

\[
\frac{a \rightarrow a'}{\text{let } x=a \text{ in } s \rightarrow \text{let } x=a' \text{ in } s} \\
\text{(MSOS-\text{LET-EXP})} \\
\frac{(\text{state}=\sigma[i/x], \text{state}'=\sigma', ...)}{s \rightarrow s'} \\
\frac{(\text{state}=\sigma, \text{state}'=\sigma'[\sigma(x)/x], ...)}{\text{let } x=i \text{ in } s \rightarrow \text{let } x=\sigma'(x) \text{ in } s'} \\
\text{(MSOS-\text{LET-STMT})} \\
\frac{\text{let } x=i \text{ in } \{ \} \rightarrow \{ \}} \\
\text{(MSOS-\text{LET-DONE})}
\]

Like for the other features, the MSOS rules are more modular than their small-step SOS variants.

Since programs are now just ordinary (closed) expressions and they are now executed in the empty state,
the rule (MSOS-\text{VAR}), namely

\[
\text{int } x; \frac{(\text{state}=x \mapsto 0, ...)}{s \rightarrow s'} \\
\]

needs to change into a rule of the from

\[
\frac{(\text{state}'=\_ , ...)}{s \rightarrow s'} \\
\]
Unfortunately, regardless of what we place instead of “?”, such a rule will not work. That is because there is nothing to prevent it to apply to any statement at any step during the reduction. To enforce it to happen only at the top of the program and only at the beginning of the reduction, we can wrap the original program (which is a statement) into a one-element configuration-like term \(\langle s \rangle\). Then the rule (MSOS-Var) can be replaced with the following rule:

\[
\langle s \rangle \xrightarrow{\text{state} = \ldots} s
\]

Putting Them All Together

The modularity of MSOS makes it quite easy to put all the features discussed above together and thus define the MSOS of IMP++. Effectively, we have to do the following:

1. We add the three label attributes used for the semantics of the individual features above, namely the read-write input attribute and the two write-only attributes output and halting.

2. We add all the MSOS rules of all the features above unchanged (nice!), except for the rule (MSOS-Var) for programs (which changed several times, anyway).

3. To initialize the label attributes, we add a pairing construct \(\langle s, \omega \rangle\) like we did when we added the input/output extension of IMP, where \(s\) is a statement (programs are ordinary statements now) and \(\omega\) is a buffer, and replace the rule (MSOS-Var) in Figure 3.24 with the following:

\[
\langle s, \omega \rangle \xrightarrow{\text{state} = x! = 0, \text{input} = \omega, \text{halting} = \text{false}, \ldots} \text{top s}
\]

It is important to note that the MSOS rules of the individual IMP extensions can be very elegantly combined into one language (IMP++). The rule (MSOS-Var) had to globally change in order to properly initialize the attribute values, but nothing had to be done in the MSOS rules of any of the features in order to put it together with the other MSOS rules of the other features.

Unfortunately, even though each individual feature has its intended semantics, the resulting IMP++ language does not. We still have the same semantic problems with regards to concurrency that we had in the context of small-step SOS in Section 3.5.6. For example, the concurrency of spawn statements is limited to the blocks in which they appear. For example, a statement of the form (\(\text{let } x = i \text{ in spawn } s_1\) \(s_2\)) does not allow \(s_2\) to be evaluated concurrently with \(s_1\). The statement \(s_1\) has to evaluate completely and then the let statement dissolved, before any step in \(s_2\) can be performed. To fix this problem, we would have to adopt a different solution, like the one proposed in Section 3.5.7 in the context of small-step SOS. Thanks to its modularity, MSOS would make such a solution easier to implement than small-step SOS. Particularly, one can use the label mechanism to pass a spawned thread and its execution environment all the way to the top modularly, without having to propagate it explicitly through language constructs.

3.6.3 MSOS in Rewrite Logic

Like big-step and small-step SOS, we can also associate a conditional rewrite rule to each MSOS rule and hereby obtain a rewrite logic theory that faithfully (i.e., step-for-step) captures the MSOS definition. There could be different ways to do this. One way to do it is to first desugar the MSOS definition into a step-for-step equivalent small-step SOS definition as discussed above, and then use the faithful embedding of small-step SOS into rewrite logic discussed in Section 3.3.3. The problem with this approach is that the resulting small-step SOS definition, and implicitly the resulting rewrite logic definition, lack the modularity of the
original MSOS definition. In other words, if one wanted to extend the MSOS definition with rules that would require global changes to its corresponding SOS definition (e.g., ones adding new semantic components into the label/configuration), then one would also need to manually incorporate all those global changes in the resulting rewrite logic definition.

We first show that any MSOS proof system, say $\mathcal{S}$, can be mechanically translated into a rewrite logic theory, say $\mathcal{R}_{\mathcal{S}}$, in such a way that two important aspects of the original MSOS definition are preserved: 1) the corresponding derivation relations are step-for-step equivalent, that is, $\mathcal{S} \vdash C \rightarrow C'$ if and only if $\mathcal{R}_{\mathcal{S}} \vdash R_C \rightarrow C'$, where $R_C$ is the corresponding syntactic translation of the MSOS sequent $C \rightarrow C'$ into a rewrite logic sequent; and 2) $\mathcal{R}_{\mathcal{S}}$ is as modular as $\mathcal{S}$. Second, we apply our generic translation technique to the MSOS formal system MSOS(IMP) defined in Section 3.6.1 and obtain a rewrite logic semantics of IMP that is step-for-step equivalent to and as modular as MSOS(IMP). The modularity of MSOS(IMP) and of $\mathcal{R}_{\mathcal{S}}$ will pay off when we extend IMP in Section 3.5. Finally, we show how $\mathcal{R}_{\mathcal{S}}$ can be seamlessly defined in Maude, yielding another interpreter for IMP (in addition to those corresponding to the big-step and small-step SOS definitions of IMP in Sections 3.2.3 and 3.3.3).

**Computationally and Modularly Faithful Embedding of MSOS into Rewrite Logic**

Our embedding of MSOS into rewrite logic is very similar to that of small-step SOS, with one important exception: the non-syntactic components of the configuration are all grouped into a record, which is a multiset of attributes, each attribute being a pair associating appropriate semantic information to a field. This allows us to use multiset matching (or matching modulo associativity, commutativity, and identity) in the corresponding rewrite rules to extract the needed semantic information from the record, thus achieving not only a computationally equivalent embedding of MSOS into rewrite logic, but also one with the same degree of modularity as MSOS.

Formally, let us assume an arbitrary MSOS formal proof system. Let $\text{Attribute}$ be a fresh sort and let $\text{Record}$ be the sort $\text{Bag}\{\text{Attribute}\}$ (that means that we assume all the infrastructure needed to define records as comma-separated bags, or multisets, of attributes). For each field $\text{Field}$ holding semantic contents $\text{Contents}$ that appears unprimed or primed in any of the labels on any of the transitions in any of the rules of the MSOS proof system, let us assume an operation “$\text{Field} = _\_ : \text{Contents} \rightarrow \text{Attribute}$” (the name of this postfix unary operation is “$\text{Field} = _\_”$). Finally, for each syntactic category $\text{Syntax}$ used in any transition that appears anywhere in the MSOS proof system, let us define a configuration construct “$_\_ _\_ : \text{Syntax} \times \text{Record} \rightarrow \text{Configuration}”$. We are now ready to define our transformation of an MSOS rule into a rewrite logic rule:

1. Translate it into an SOS rule, as discussed right above Section 3.6.1; we could also go directly from MSOS rules to rewrite logic rules, but we would have to repeat most of the steps from MSOS to SOS that were already discussed;
2. Group all the semantic components in the resulting SOS configurations into a corresponding record, where each semantic component translates into a corresponding attribute using a corresponding label;
3. Replace those attributes whose semantic components are not used anywhere in the rule by a generic variable of sort $\text{Record}$;
4. Finally, use the technique in Section 3.3.3 to transform the resulting SOS-like rules into rewrite rules, tagging the left-hand-side configurations with the $\circ$ symbol.
Applying the steps above, the four MSOS rules discussed right above Section [3.6.1] (namely the ones for variable assignment, printing to the output, sequential composition of statements, and variable lookup) translate into the following rewrite logic rules:

\[
\begin{align*}
&\circ(X = I, (\text{state} = \sigma, \rho)) \rightarrow \langle \langle \rangle, (\text{state} = \sigma[I/X], \rho) \rangle \\
&\circ(\text{print}(I); (\text{output} = \omega, \rho)) \rightarrow \langle \langle \rangle, (\text{output} = \omega : I, \rho) \rangle \\
&\quad\quad \text{if } \circ(S_1, S_2, \rho) \rightarrow \langle S_1 ', S_2, \rho' \rangle \\
&\circ(X, (\text{state} = \sigma, \rho)) \rightarrow \langle \sigma(X), (\text{state} = \sigma, \rho) \rangle \\
\end{align*}
\]

We use the same mechanism as for small-step SOS to obtain the reflexive and transitive many-step closure of the MSOS one-step transition relation. This mechanism was discussed in detail in Section [3.3.3]; it essentially consists of adding a configuration marker \(\star\) together with a rule "\(\star Cfg \rightarrow \star Cfg\)" if \(\circ Cfg \rightarrow Cfg\)" iteratively applying the one-step relation.

**Theorem 16. (Faithful embedding of MSOS into rewrite logic)** For any MSOS definition MSOS, and any appropriate configurations \(C\) and \(C'\), the following equivalences hold:

\[
\begin{align*}
\text{MSOS} \vdash C \rightarrow C' &\iff R_{\text{MSOS}} \vdash \circ C \rightarrow^1 \overline{C'} \iff R_{\text{MSOS}} \vdash \circ C \rightarrow C' \\
\text{MSOS} \vdash C \rightarrow^* C' &\iff R_{\text{MSOS}} \vdash \star C \rightarrow \star C' \\
\end{align*}
\]

where \(R_{\text{MSOS}}\) is the rewrite logic semantic definition obtained from MSOS by translating each rule in MSOS as above. (Recall from Section [2.5] that \(\rightarrow^1\) is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewrite logic. Also, as \(C\) and \(C'\) are parameter-free—parameters only appear in rules—, \(\overline{C}\) and \(\overline{C'}\) are ground terms.)

Like for the previous embeddings of big-step and small-step SOS into rewrite logic, let us elaborate on the apparent differences between MSOS and \(R_{\text{MSOS}}\) from a user perspective. The most visible difference is the SOS-like style of writing the rules, namely using configurations instead of labels, which also led to the inheritance of the \(\circ\) mechanism from the embedding of SOS into rewrite logic. Therefore, the equivalent rewrite logic definition is slightly more verbose than the original MSOS definition. On the other hand, it has the advantage that it is more direct than the MSOS definition, in that it eliminates all the notational conventions. Indeed, if we strip MSOS out of its notational conventions and go straight to its essence, we find that that essence is precisely its use of multiset matching to modularly access the semantic components of the configuration. MSOS chose to do this on the labels, using specialized conventions for read-only/write-only/read-write components, while our rewrite logic embedding of MSOS does it in the configurations, uniformly for all semantic components. Where precisely this matching takes place is, in our view, less relevant. What is relevant and brings MSOS its modularity is that multiset matching does happen. Therefore, similarly to the big-step and small-step SOS representations in rewrite logic, we conclude that the rewrite theory \(R_{\text{MSOS}}\) is MSOS, and not an encoding of it.

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, unfortunately, \(R_{\text{MSOS}}\) (and implicitly MSOS) still lacks the main strengths of rewrite logic, namely context-insensitive and parallel application of rewrite rules. Indeed, the rules of \(R_{\text{MSOS}}\) can only apply at the top, sequentially, so these rewrite theories corresponding to the faithful embedding of MSOS follow a rather poor rewrite logic style. Like for the previous embeddings, this is not surprising though and does not question the quality of our embeddings. All it says is that MSOS was not meant to have the capabilities of rewrite logic with regards to context-insensitivity and parallelism; indeed, all MSOS attempts to achieve is to address the lack of modularity of SOS and we believe that it succeeded in doing so. Unfortunately, SOS has several other major problems, which are discussed in Section [3.3].
MSOS of IMP in Rewrite Logic

We here discuss the complete MSOS definition of IMP in rewrite logic, obtained by applying the faithful embedding technique discussed above to the MSOS definition of IMP in Section 3.6.1. Figure 3.25 gives an algebraic definition of configurations as well as needed additional record infrastructure; all the sorts, operations, and rules in Figure 3.25 were already discussed either above or in Section 3.3.3. Figure 3.26 gives the rules of the rewrite logic theory RMOS(IMP), that is obtained by applying the procedure above to the MSOS of IMP in Figures 3.23 and 3.24. Like before, we used the rewrite logic convention that variables start with upper-case letters; if they are greek letters, then we use a similar but larger symbol (e.g., \( \sigma \) instead of \( \sigma \) for variables of sort State, or \( \rho \) instead of \( \rho \) for variables of sort Record). The following corollary of Theorem 16 establishes the faithfulness of the representation of the MSOS of IMP in rewrite logic:

**Corollary 6.** \( \text{MSOS}(\text{IMP}) \vdash C \rightarrow C' \iff \text{RMOS}((\text{IMP}) \vdash \text{C} \rightarrow \text{C}' \).**

Therefore, there is no perceivable computational difference between the IMP-specific proof system MSOS(IMP) and generic rewrite logic deduction using the IMP-specific rewrite rules in RMOS(IMP); the two are faithfully equivalent. Moreover, by the discussion following Theorem 16, RMOS(IMP) is also as modular as MSOS(IMP). This will be further emphasized in Section 3.5 where we will extend IMP with several features, some of which requiring more attributes.

★ Maude Definition of IMP MSOS

Figure 3.27 shows a straightforward Maude representation of the rewrite theory RMOS(IMP) in Figures 3.26 and 3.25. The Maude module IMP-SEMANTICS-MSOS in Figure 3.27 is executable, so Maude, through its rewriting capabilities, yields an MSOS interpreter for IMP the same way it yielded big-step and small-step SOS interpreters in Sections 3.2.3 and 3.3.3 respectively; for example, the Maude rewrite command

```
rewrite * < sumPgm > .
```

where sumPgm is the first program defined in the module IMP-PROGRAMS in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by “…”):

```
227
```
\( \circ (X, (\text{state} = \sigma, \rho)) \rightarrow \langle \sigma(X), (\text{state} = \sigma, \rho) \rangle \) if \( \sigma(X) \neq \bot \)

\( \circ (A_1 + A_2, \rho) \rightarrow \langle A'_1 + A'_2, \rho' \rangle \) if \( \circ (A_1, \rho) \rightarrow \langle A'_1, \rho' \rangle \)

\( \circ (A_1 + A_2, \rho) \rightarrow \langle A_1 + A'_2, \rho' \rangle \) if \( \circ (A_2, \rho) \rightarrow \langle A'_2, \rho' \rangle \)

\( \circ (I_1 + I_2, \rho) \rightarrow \langle I_1 + I_2, \rho \rangle \)

\( \circ (A_1 / A_2, \rho) \rightarrow \langle A'_1 / A'_2, \rho' \rangle \) if \( \circ (A_1, \rho) \rightarrow \langle A'_1, \rho' \rangle \)

\( \circ (A_1 / A_2, \rho) \rightarrow \langle A_1 / A'_2, \rho' \rangle \) if \( \circ (A_2, \rho) \rightarrow \langle A'_2, \rho' \rangle \)

\( \circ (I_1 / I_2, \rho) \rightarrow \langle I_1 / I_2, \rho \rangle \) if \( I_2 \neq 0 \)

\( \circ (A_1 <= A_2, \rho) \rightarrow \langle A'_1 <= A'_2, \rho' \rangle \) if \( \circ (A_1, \rho) \rightarrow \langle A'_1, \rho' \rangle \)

\( \circ (I_1 <= A_2, \rho) \rightarrow \langle I_1 <= A'_2, \rho' \rangle \) if \( \circ (A_2, \rho) \rightarrow \langle A'_2, \rho' \rangle \)

\( \circ (I_1 <= I_2, \rho) \rightarrow \langle I_1 \leq I_2, \rho \rangle \)

\( \circ (! B, \rho) \rightarrow \langle ! B', \rho' \rangle \) if \( \circ (B, \rho) \rightarrow \langle B', \rho' \rangle \)

\( \circ (! \text{true}, \rho) \rightarrow \langle \text{false}, \rho \rangle \)

\( \circ (! \text{false}, \rho) \rightarrow \langle \text{true}, \rho \rangle \)

\( \circ (B_1 && B_2, \rho) \rightarrow \langle B'_1 && B'_2, \rho' \rangle \) if \( \circ (B_1, \rho) \rightarrow \langle B'_1, \rho' \rangle \)

\( \circ (\text{false} && B_2, \rho) \rightarrow \langle \text{false}, \rho \rangle \)

\( \circ (\text{true} && B_2, \rho) \rightarrow \langle B_2, \rho \rangle \)

\( \circ ([S], \rho) \rightarrow \langle S, \rho \rangle \)

\( \circ (X = A ;, \rho) \rightarrow \langle X = A', ;, \rho' \rangle \) if \( \circ (A, \rho) \rightarrow \langle A', \rho' \rangle \)

\( \circ (X = I ;, (\text{state} = \sigma, \rho)) \rightarrow \langle [], (\text{state} = \sigma[I/X], \rho) \rangle \) if \( \sigma(X) \neq \bot \)

\( \circ (S_1 S_2, \rho) \rightarrow \langle S'_1 S'_2, \rho' \rangle \) if \( \circ (S_1, \rho) \rightarrow \langle S'_1, \rho' \rangle \)

\( \circ ([] S_2, \rho) \rightarrow \langle S_2, \rho \rangle \)

\( \circ (\text{if} (B) S_1 \text{else} S_2, \rho) \rightarrow \langle \text{if} (B') S_1 \text{else} S_2, \rho' \rangle \) if \( \circ (B, \rho) \rightarrow \langle B', \rho' \rangle \)

\( \circ (\text{if} (\text{true}) S_1 \text{else} S_2, \rho) \rightarrow \langle S_1, \rho \rangle \)

\( \circ (\text{if} (\text{false}) S_1 \text{else} S_2, \rho) \rightarrow \langle S_2, \rho \rangle \)

\( \circ (\text{while} (B) S, \rho) \rightarrow \langle \text{if} (B) \{S \text{ while } (B) S \} \text{ else } [], \rho \rangle \)

\( \circ (\text{int } X I; S) \rightarrow \langle S, (\text{state} = X I \mapsto 0) \rangle \)

Figure 3.26: \( R_{\text{MSOS}}^{\text{IMP}} \): the complete MSOS of IMP in rewrite logic.
mod IMP-CONFIGURATIONS-MSOS is including IMP-SYNTAX + STATE.
sorts Attribute Record Configuration ExtendedConfiguration.
subsort Attribute < Record.
subsort Configuration < ExtendedConfiguration.
op empty : -> Record.
op _=_ : Record Record -> Record [assoc comm id: empty].
op state'_=_ : State -> Attribute.
op <,>_ : AEExp Record -> Configuration.
op <,>_ : BEExp Record -> Configuration.
op <,>_ : Stmt Record -> Configuration.
op < > : Pgm -> Configuration.
op o_ : Configuration -> ExtendedConfiguration [prec 80]. --- one step
op *_ : Configuration -> ExtendedConfiguration [prec 80]. --- all steps
var Cfg Cfg' : Configuration.
crl * Cfg => * Cfg' if o Cfg => Cfg'.
endm

mod IMP-SEMANTICS-MSOS is including IMP-CONFIGURATIONS-MSOS.
var X : Id . var R R' : Record . var Sigma Sigma' : State .
var I I1 I2 : Int . var Xl : List{Id} . var S S1 S1' S2 : Stmt .
crl o < X,(state = Sigma, R) > => < Sigma(X),(state = Sigma, R) >
    if Sigma(X) /==Bool undefined.
crl o < A1 + A2,R > => < A1' + A2,R' > if o < A1,R > => < A1',R' >.
rl o < I1 + I2,R > => < I1 +Int I2,R >.
crl o < A1 / A2,R > => < A1'/ A2,R' > if o < A1,R > => < A1',R' >.
crl o < A1 / A2,R > => < A1 / A2',R' > if o < A2,R > => < A2',R' >.
crl o < I1 / I2,R > => < I1 /Int I2,R > if I2 =/=Bool 0.
crl o < A1 <= A2,R > => < A1 <= A2',R' > if o < A2,R > => < A2',R' >.
rl o < I1 <= I2,R > => < I1 <=Int I2,R >.
crl o < ! B,R > => < ! B,R' > if o < B,R > => < B',R' >.
rl o < ! true,R > => < false,R >.
rl o < ! false,R > => < true,R >.
rl o < true && B2,R > => < false,R >.
rl o < false && B2,R > => < false,R >.
rl o < {S}, R > => < S,R >.
crl o < X = A ;,R > => < X = A ;,R' > if o < A,R > => < A',R' >.
crl o < X = I ;,(state = Sigma, R) > => < {},(state = Sigma[I / X], R) >
    if Sigma(X) /==Bool undefined.
crl o < S1 S2,R > => < S1' S2,R' > if o < S1,R > => < S1',R' >.
rl o < {} S2,R > => < S2,R >.
crl o < if (B) S1 else S2,R > => < if (B') S1 else S2,R' > if o < B,R > => < B',R' >.
rl o < if (true) S1 else S2,R > => < S1,R >.
rl o < if (false) S1 else S2,R > => < S2,R >.
rl o < while (B) S,R > => < if (B) {S while (B) S} else {},R >.
rl o < int Xl ; S > => < S,(state = Xl -> 0) >.
endm

Figure 3.27: The MSOS of IMP in Maude, including the definition of configurations.
rewrites: 7632 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < {}, state = (n |-> 0 & s |-> 5050) >

Note that the rewrite command above took the same number of rewrite steps as the similar command executed on the small-step SOS of IMP in Maude discussed in Section 3.3.3 namely 7632. This is not unexpected, because matching is not counted as rewrite steps, no matter how complex it is.

Like for the big-step and small-step SOS definitions in Maude, one can also use any of the general-purpose tools provided by Maude on the MSOS definition above. For example, one can exhaustively search for all possible behaviors of a program using the search command:

    search * < sumPgm > =>! Cfg:ExtendedConfiguration .

As expected, only one behavior will be discovered because our IMP language so far is deterministic. The same number of states as in the case of small-step SOS will be generated by this search command, 1709.

### 3.6.4 Notes

Modular Structural Operational Semantics (MSOS) was introduced in 1999 by Mosses [51] and since then mainly developed by himself and his collaborators (e.g., [52, 53, 54]). In this section we used the implicitly-modular variant of MSOS introduced in [54], which, as acknowledged by the authors of [54], was partly inspired from discussions with us. To be more precise, we used a slightly simplified version of implicitly-modular MSOS here. In MSOS in its full generality, one can also declare some transitions unobservable; to keep the presentation simpler, we here omitted all the observability aspects of MSOS.

The idea of our representation of MSOS into rewrite logic adopted in this section is taken over from Serbanuta et al. [74]. At our knowledge, Meseguer and Braga [44] give the first representation of MSOS into rewrite logic. The representation in [44] also led to the development of the Maude MSOS tool [16], which was the core of Braga’s doctoral thesis. What is different in the representation of Meseguer and Braga in [44] from ours is that the former uses two different types of configuration wrappers, one for the left-hand-side of the transitions and one for the right-hand-side; this was already discussed in Section 3.3.4.

### 3.6.5 Exercises

**Exercise 120.** Redo all the exercises in Section 3.3.5 but for the MSOS of IMP discussed in Section 3.6.1 instead of its small-step SOS in Section 3.3.2. Skip Exercises 68, 69 and 76 since the SOS proof system there drops the syntactic components in the RHS configurations in transitions, making it unsuitable for MSOS. For the MSOS variant of Exercise 70 just follow the same non-modular approach as in the case of SOS and not the modular MSOS approach discussed in Section 3.6.2 (Exercise 123 addresses the modular MSOS variant of abrupt termination).

**Exercise 121.** Same as Exercise 86 but for MSOS instead of small-step SOS: add variable increment to IMP, like in Section 3.6.2.

**Exercise 122.** Same as Exercise 90 but for MSOS instead of small-step SOS: add input/output to IMP, like in Section 3.6.2.

**Exercise 123.** Same as Exercise 95 but for MSOS instead of small-step SOS: add abrupt termination to IMP, like in Section 3.6.2.

---

8In fact, drafts of this book preceding [54] had already dropped the implicit labels in MSOS rules, for notational simplicity.
Exercise 124. Same as Exercise 104 but for MSOS instead of small-step SOS: add dynamic threads to IMP, like in Section 3.6.2

Exercise 125. Same as Exercise 109 but for MSOS instead of small-step SOS: add local variables using let to IMP, like in Section 3.6.2

Exercise 126. This exercise asks to define IMP++ in MSOS, in various ways. Specifically, redo Exercises 114, 115, 116, 117, and 118 but for the MSOS of IMP++ discussed in Section 3.6.2 instead of its small-step SOS in Section 3.5.6
look into ThinkerType

should I add more stuff from [74], in particular the more precise embedding transformation from there? i mean, is the transformation discussed here detailed enough?

should I say more about [44]?
3.7 Reduction Semantics with Evaluation Contexts

The small-step SOS/MSOS approaches discussed in Sections 3.3 and 3.6 define a language semantics as a proof system whose rules are mostly conditional. The conditions of such rules allow to implicitly capture the program execution context as a proof context. This shift of focus from the informal notion of execution context to the formal notion of proof context has a series of advantages and it was, in fact, the actual point of formalizing language semantics using SOS. However, as the complexity of programming languages increased, in particular with the adoption of control-intensive statements like call/cc (e.g., Scheme) that can arbitrarily change the execution context, the need for an explicit representation of the execution context as a first-class citizen in the language semantics also increased. Reduction semantics with evaluation contexts (RSEC) is a variant of small-step SOS where the evaluation context may appear explicit in the term being reduced.

In an RSEC language definition one starts by defining the syntax of evaluation contexts, or simply just contexts, which is typically done by means of a context-free grammar (CFG). A context is a program or a fragment of program with a hole, where the hole, which is written □, is a placeholder for where the next computational step can take place. If \( c \) is an evaluation context and \( e \) is some well-formed appropriate fragment (expression, statement, etc.), then \( c[e] \) is the program or fragment obtained by replacing the hole of \( c \) by \( e \). Reduction semantics with evaluation contexts relies on a tacitly assumed (but rather advanced) parsing mechanism that takes a program or a fragment \( p \) and decomposes it into a context \( c \) and a subprogram or fragment \( e \), called a redex, such that \( p = c[e] \). This decomposition process is called splitting (of \( p \) into \( c \) and \( e \)). The inverse process, composing a redex \( e \) and a context \( c \) into a program or fragment \( p \), is called plugging (of \( e \) into \( c \)). These operations are assumed whenever needed.

Consider a language with arithmetic/Boolean expressions and statements like our IMP language in Section 3.1. A possible CFG definition of evaluation contexts for such a language may include the following productions (the complete definition of IMP evaluation contexts is given in Figure 3.30):

\[
\text{Context} ::= \quad □ \\
\quad \mid \quad \text{Context} \Leftarrow \text{AExp} \\
\quad \mid \quad \text{Int} \Leftarrow \text{Context} \\
\quad \mid \quad \text{Id} = \text{Context} ; \\
\quad \mid \quad \text{Context Stmt} \\
\quad \mid \quad \text{if (Context) Stmt else Stmt} \\
\quad \mid \quad \ldots
\]

Note how the intended evaluation strategies of the various language constructs are reflected in the definition of evaluation contexts: \( \Leftarrow \) is sequentially strict (\( □ \) allowed to go into the first subexpression until evaluated to an \( \text{Int} \), then into the second subexpression), the assignment is strict only in its second argument, while the sequential composition and the conditional are strict only in their first arguments. If one thinks of language constructs as operations taking a certain number of arguments of certain types, then note that the operations appearing in the grammar defining evaluation contexts are different from their corresponding operations.
appearing in the language syntax; for example, “\( Id = \text{Context}; \)” is different from “\( Id = AExp; \)” because the former takes a context as second argument while the latter takes an arithmetic expression.

Here are some examples of correct evaluation contexts for the grammar above:

\[
\begin{align*}
\square & \\
3 \leq \square & \\
\square & \leq 3 \\
\square & \ x = 5;, \text{ where } x \text{ is any variable.}
\end{align*}
\]

\[\text{if } (\square) \ s_1 \ \text{else } s_2, \text{ where } s_1 \text{ and } s_2 \text{ are any well-formed statements.}\]

Here are some examples of incorrect evaluation contexts:

\[
\begin{align*}
\square & \leq \square — \text{a context can have only one hole.} \\
x & \leq 3 — \text{a context must contain a hole.} \\
x & \leq \square — \text{the first argument of } \leq \text{ must be an integer number in order to allow the hole in the second argument.} \\
x & = 5; \square — \text{the hole can only appear in the first statement in a sequential composition.} \\
\square & = 5; — \text{the hole cannot appear as first argument of the assignment construct.} \\
\text{if } (x \leq 7) \square \ \text{else } x = 5; — \text{the hole is only allowed in the condition of a conditional.}
\end{align*}
\]

Here are some examples of decompositions of syntactic terms into a context and a redex (recall that we can freely use parentheses for disambiguation; here we enclose evaluation contexts in parentheses for clarity):

\[
\begin{align*}
7 & = (\square)[7] \\
3 \leq x & = (3 \leq \square)[x] = (\square \leq x)[3] = (\square)[3 \leq x] \\
3 \leq (2 + x) + 7 & = (3 \leq \square + 7)[2 + x] = (\square \leq (2 + x) + 7)[3] = \ldots
\end{align*}
\]

For simplicity, we consider only one type of context in this section, but in general one can have various types, depending upon the types of their holes and of their result.

Reduction semantics with evaluation contexts tends to be a purely syntactic definitional framework (following the slogan “everything is syntax”). If semantic components are necessary in a particular definition, then they are typically “swallowed by the syntax”. For example, if one needs a state as part of the configuration for a particular language definition (like we need for our hypothetical \( \text{IMP} \) language discussed here), then one adds a context production of the form

\[
\text{Context} \ ::= \ (\text{Context}, \text{State})
\]

where the \( \text{State} \), an inherently semantic entity, becomes part of the evaluation context. Note that once one adds additional syntax to evaluation contexts that does not correspond to constructs in the syntax of the original language, such as our pairing of a context and a state above, one needs to also extend the original syntax with corresponding constructs, so that the parsing-like mechanism decomposing a syntactic term into a context and a redex can be applied. In our case, the production above suggests that a pairing configuration construct of the form \( \langle \text{Stmt}, \text{State} \rangle \), like for SOS, also needs to be defined. Unlike in SOS, we do not need configurations pairing other syntactic categories with a state, such as \( \langle AExp, \text{State} \rangle \) and \( \langle BExp, \text{State} \rangle \); the
When this rule is applied, we say that $e \rightarrow e'$ where $e$ is any appropriate evaluation context (i.e., such that $c[e]$ is well-formed programs or fragments of program). This rule is called the characteristic rule of RSEC. When this rule is applied, we say that $e$ reduces to $e'$ in context $c$. If $c$ is the empty context $\Box$ then $c[e]$ is $e$ and thus the characteristic rule is useless; for that reason, the characteristic rule may be encountered with a side condition “if $c \neq \Box$”. Choosing good strategies to search for splits of terms into contextual representations can be a key factor in obtaining efficient implementations of RSEC execution engines.

The introduction of the characteristic rule allows us to define reduction semantics of languages or calculi quite compactly. For example, here are all the rules needed to completely define the semantics of the comparison, sequential composition and conditional language constructs for which we defined evaluation contexts above:

$$i_1 \Leftarrow i_2 \rightarrow i_1 \leq_{hat} i_2$$

$$\begin{array}{l}
| \text{if (true)} \, \text{s}_1 \text{else} \, \text{s}_2 \rightarrow \text{s}_1 \\
| \text{if (false)} \, \text{s}_1 \text{else} \, \text{s}_2 \rightarrow \text{s}_2 \\
\end{array}$$

The characteristic rule tends to be the only conditional rule in an RSEC, in the sense that the remaining rules take no reduction premises (though they may still have side conditions). Moreover, as already pointed out, the characteristic rule is actually unnecessary, because one can very well replace each rule $l \rightarrow r$ by a rule $c[l] \rightarrow c[r]$. The essence of reduction semantics with evaluation contexts is not its characteristic reduction rule, but its specific approach to defining evaluation contexts as a grammar and then using them as an explicit part of languages or calculi definitions. The characteristic reduction rule can therefore be regarded as “syntactic sugar”, or convenience to the designer allowing her to write more compact definitions.
To give the semantics of certain language constructs, one may need to access specific information that is stored inside an evaluation context. For example, consider a term \( x \leq 3, (x \mapsto 1, y \mapsto 2) \), which can be split as \( c[x] \), where \( c \) is the context \( \langle \square \leq 3, (x \mapsto 1, y \mapsto 2) \rangle \). In order to reduce \( c[x] \) to \( c[1] \) as desired, we need to look inside \( c \) and find out that the value of \( x \) in the state held by \( c \) is 1. Therefore, following the purely syntactic style adopted so far in this section, the reduction semantics with evaluation contexts rule for variable lookup in our case here is the following:

\[
\langle c, \sigma \rangle[x] \rightarrow \langle c, \sigma(\sigma(x)) \rangle \quad \text{if } \sigma(x) \neq \bot
\]

Indeed, the same way we add as much structure as needed in ordinary terms, we can add as much structure as needed in evaluation contexts. Similarly, below is the rule for variable assignment:

\[
\langle c, \sigma \rangle[x = i;] \rightarrow \langle c, \sigma[i/x]\{\} \rangle \quad \text{if } \sigma(x) \neq \bot
\]

Note that in this case both the context and the redex were changed by the rule. In fact, as discussed in Section 3.10, one of the major benefits of reduction semantics with evaluation contexts consists in precisely the fact that one can arbitrarily modify the evaluation context in rules; this is crucial for giving semantics to control-intensive language constructs such as call/cc.

Splitting of a term into an evaluation context and a redex does not necessarily need to take place at the top of the left-hand-side of a rule. For example, the following is an alternative way to give reduction semantics with evaluation contexts to variable lookup and assignment:

\[
\langle c[x], \sigma \rangle \rightarrow \langle c[\sigma(x)], \sigma \rangle \quad \text{if } \sigma(x) \neq \bot
\]
\[
\langle c[x = i;], \sigma \rangle \rightarrow \langle c[\{\}], \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \bot
\]

Note that, even if we follow this alternative style, we still need to include the production \( \text{Context} ::= \langle \text{Context}, \text{State} \rangle \) to the evaluation context CFG if we want to write rules as \( c[i_1 \leq i_2] \rightarrow \langle c[1 \leq \text{Int}_{\text{k}} i_2] \rangle \) or to further take advantage of the characteristic rule and write elegant and compact rules such as \( i_1 \leq i_2 \rightarrow i_1 \leq \text{Int}_{\text{k}} i_2 \). If we want to completely drop evaluation context productions that mix syntactic and semantic components, such as \( \text{Context} ::= \langle \text{Context}, \text{State} \rangle \), then we may adopt one of the styles discussed in Exercises 127 and 128, respectively, though one should be aware of the fact that those styles also have their disadvantages.

Figure 3.29 shows a reduction sequence using the evaluation contexts and the rules discussed so far. We used the following (rather standard) notation for instantiated contexts whenever we applied the characteristic rule: the redex is placed in a box replacing the hole of the context. For example, the fact that expression \( 3 \leq x \) is split into contextual representation \( (3 \leq \square)[x] \) is written compactly and intuitively as \( 3 \leq [x] \). Note that the evaluation context changes almost at each step during the reduction sequence in Figure 3.29.

Like in small-step SOS and MSOS, we can also transitively and reflexively close the one-step transition relation \( \rightarrow \). As usual, we let \( \rightarrow^* \) denote the resulting multi-step transition relation.

### 3.7.1 The Reduction Semantics with Evaluation Contexts of IMP

Figure 3.30 shows the definition of evaluation contexts for IMP and Figure 3.31 shows all the reduction semantics rules of IMP using the evaluation contexts defined in Figure 3.30. The evaluation context productions capture the intended evaluation strategies of the various language constructs. For example, + and / are non-deterministically strict, so any one of their arguments can be reduced one step whenever the sum or the division expression can be reduced one step, respectively, so the hole \( \square \) can go in any of their two subexpressions. As previously discussed, in the case of \( \triangleleft \) one can reduce its second argument only after its first argument is fully reduced (to an integer). The evaluation strategy of \( ! \) is straightforward. For \&\&, note
\[
\begin{align*}
\langle x = 1; y = 2; \text{if} (x \leq y) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 0, y \mapsto 0) \\
\rightarrow \langle \{ \} \ y = 2; \text{if} (x \leq y) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 1, y \mapsto 0) \\
\rightarrow \langle y = 2; \text{if} (x \leq y) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 1, y \mapsto 0) \\
\rightarrow \langle \{ \} \text{if} (x \leq y) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 1, y \mapsto 2) \\
\rightarrow \langle \text{if} (x \leq y) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 1, y \mapsto 2) \\
\rightarrow \langle \text{if} (y \leq y) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 1, y \mapsto 2) \\
\rightarrow \langle \text{if} (1 \leq y) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 1, y \mapsto 2) \\
\rightarrow \langle \text{if} (\text{true}) \{ x = 0; \} \text{else} \{ y = 0; \}, (x \mapsto 1, y \mapsto 2) \\
\rightarrow \langle x = 0; \}, (x \mapsto 1, y \mapsto 2) \\
\rightarrow \langle \{ \}, (x \mapsto 0, y \mapsto 2) 
\end{align*}
\]

Figure 3.29: Sample reduction sequence.

<table>
<thead>
<tr>
<th>IMP evaluation contexts syntax</th>
<th>IMP language syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context</strong> ::= □</td>
<td><strong>AExp</strong> ::= <strong>Int</strong></td>
</tr>
<tr>
<td><strong>Context</strong> + <strong>AExp</strong></td>
<td><strong>AExp</strong> + <strong>AExp</strong></td>
</tr>
<tr>
<td></td>
<td><strong>AExp</strong> / <strong>AExp</strong></td>
</tr>
<tr>
<td><strong>Context</strong> / <strong>AExp</strong></td>
<td><strong>BExp</strong> ::= <strong>Bool</strong></td>
</tr>
<tr>
<td></td>
<td><strong>AExp</strong> &lt;= <strong>AExp</strong></td>
</tr>
<tr>
<td></td>
<td><strong>! BExp</strong></td>
</tr>
<tr>
<td></td>
<td><strong>BExp</strong> &amp; &amp; <strong>BExp</strong></td>
</tr>
<tr>
<td><strong>Context</strong> &lt;= <strong>AExp</strong></td>
<td><strong>Block</strong> ::= <strong>{}</strong></td>
</tr>
<tr>
<td><strong>Int</strong> &lt;= <strong>Context</strong></td>
<td><strong>Stmt</strong> ::= <strong>Block</strong></td>
</tr>
<tr>
<td><strong>! Context</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Context</strong> &amp; &amp; <strong>BExp</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Id</strong> = <strong>Context</strong>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Context</strong> <strong>Stmt</strong></td>
</tr>
<tr>
<td><strong>if</strong> (<strong>Context</strong>) <strong>Stmt</strong> <strong>else</strong> <strong>Stmt</strong></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.30: Evaluation contexts for IMP (left column); the syntax of IMP (from Figure 3.1) is recalled in the right column only for reader’s convenience, to more easily compare the two grammars.
that only its first argument is reduced. Indeed, recall that && has a short-circuited semantics, so its second argument is reduced only after the first one is completely reduced (to a Boolean) and only if needed; this is defined using rules in Figure 3.31. The evaluation contexts for assignment, sequential composition, and the conditional have already been discussed.

Many of the rules in Figure 3.31 have already been discussed or are trivial. Note that there is no production Context ::= while (Context) Stmt as a hasty reader may (mistakenly) expect. That is because such a production would allow the evaluation of the Boolean expression in the while loop’s condition to a Boolean value in the current context; supposing that value is true, then, unless one modifies the syntax in some rather awkward way, we cannot recover the original Boolean expression to evaluate it again after the evaluation of the while loop’s body statement. The solution to handle loops remains the same as in SOS, namely to explicitly unroll them into conditional statements, as shown in Figure 3.31. Note that the evaluation contexts allow the loop unrolling to only happen when the while statement is a redex. In particular, after an unrolling reduction takes place, subsequent unrolling steps are disallowed inside the then branch; to unroll it again, the loop statement must become again a redex, which can only happen after the conditional statement is itself reduced. The initial program configuration (containing only the program) is reduced also like in SOS (last rule in Figure 3.31) note that one cannot instead define a production Context ::= int List(Id); Context, because, for example, there is no way to reduce “x = 5;” in ⟨int x; [x = 5;].

238
3.7.2 The Reduction Semantics with Evaluation Contexts of IMP++

We next discuss the reduction semantics of IMP++ using evaluation contexts. Like for the other semantics, we first add each feature separately to IMP and then we add all of them together and investigate the modularity and appropriateness of the resulting definition.

Variable Increment

The use of evaluation contexts makes the definition of variable increment quite elegant and modular:

\[ \langle c, \sigma \rangle[\text{++ } x] \rightarrow \langle c, \sigma[\sigma(x) + \text{Int } 1]/x \rangle[\sigma(x) + \text{Int } 1] \]

No other rule needs to change, because: (1) unlike in SOS, all the language-specific rules are unconditional, each rule matching and modifying only its relevant part of the configuration; and (2) the language-independent characteristic rule allows reductions to match and modify only their relevant part of the configuration, propagating everything else in the configuration automatically.

Input/Output

We need to first change the configuration employed by our reduction semantics with evaluation contexts of IMP from \( \langle s, \sigma \rangle \) to \( \langle s, \sigma, \omega_{\text{in}}, \omega_{\text{out}} \rangle \), to also include the input/output buffers. This change, unfortunately, generates several other changes in the existing semantics, some of them non-modular in nature. First, we need to change the syntax of (statement) configurations to include input and output buffers, and the syntax of contexts from `Context ::= ... | (Context, State) to Context ::= ... | (Context, State, Buffer, Buffer)`. No matter what semantic approach one employs, some changes in the configuration (or its equivalent) are unavoidable when one adds new language features that require new semantic data, like input/output constructs that require their own buffers (recall that, e.g., in MSOS, we had to add new attributes in transition labels instead). Hence, this change is acceptable. What is inconvenient (and non-modular), however, is that the rules for variable lookup and for assignment need the complete configuration, so they have to change from

\[ \langle c, \sigma \rangle[x] \rightarrow \langle c, \sigma[\sigma(x)] \rangle \]
\[ \langle c, \sigma \rangle[x = i;] \rightarrow \langle c, \sigma[i/x] \rangle[\{\}]] \]

\[ \langle c, \sigma, \omega_{\text{in}}, \omega_{\text{out}} \rangle[x] \rightarrow \langle c, \sigma, \omega_{\text{in}}, \omega_{\text{out}}[\sigma(x)] \rangle \]
\[ \langle c, \sigma, \omega_{\text{in}}, \omega_{\text{out}}[x = i;] \rightarrow \langle c, \sigma[i/x], \omega_{\text{in}}, \omega_{\text{out}}[\{\}] \rangle \]

Also, the initial configuration which previously held only the program, now has to change to hold both the program and an input buffer, and the rule for programs (the last in Figure 3.31) needs to change as follows:

\[ \langle \text{int } x l; \ s, \omega_{\text{in}} \rangle \rightarrow \langle s, (x l \mapsto 0), \omega_{\text{in}}, \epsilon \rangle \]

Once the changes above are applied, we are ready for adding the evaluation context for the print statement as well as the reduction semantics rules of both input/output constructs:

\[ Context ::= ... | \text{print}(\text{Context}); \]
\[ \langle c, \sigma, i : \omega_{\text{in}}, \omega_{\text{out}}[\text{read()}] \rightarrow \langle c, \sigma, \omega_{\text{in}}, \omega_{\text{out}}[\{i\}] \rangle \]
\[ \langle c, \sigma, \omega_{\text{in}}, \omega_{\text{out}}[\text{print}(i);] \rightarrow \langle c, \sigma, \omega_{\text{in}}, \omega_{\text{out}} : i[\{\}] \rangle \]

Other possibilities to add input/output buffers to the configuration and to give the reduction semantics with evaluation contexts of the above language features in a way that appears to be more modular (but which yields other problems) are discussed in Section 3.10.
Abrupt Termination

For our language, reduction semantics allows very elegant, natural and modular definitions of abrupt termination, without having to extent the syntax of the original language and without adding any new reduction steps as an artifact of the approach chosen:

\[
\langle c, \sigma \rangle \left[ \text{halt;} \right] \rightarrow \langle \{\}, \sigma \rangle
\]

Therefore, the particular evaluation context in which the abrupt termination is being generated, \( c \), is simply discarded. This is not possible in any of the big-step, small-step or MSOS styles above, because in there the evaluation context \( c \) is captured by the proof context, which, like in any logical system, cannot be simply discarded. The elegance of the two rules above suggests that having the possibility to explicitly match and change the evaluation context is a very powerful and convenient feature of a language semantic framework.

Dynamic Threads

The rules (\textsc{SmallStep-Spawn-Arg}) (resp. (\textsc{MSOS-Spawn-Arg})) and (\textsc{SmallStep-Spawn-Wait}) (resp. (\textsc{MSOS-Spawn-Wait})) in Section \textsection3.5.4 (resp. Section \textsection3.6.2) are essentially computation propagation rules. In reduction semantics with evaluation contexts the role of such rules is taken over by the splitting/plugging mechanism, which in turn relies on parsing and therefore needs productions for evaluation contexts. We can therefore replace those rules by appropriate productions for evaluation contexts:

\[
\text{Context ::= ...} \\
| \text{spawn Context} \\
| \text{spawn Stmt Context}
\]

The second evaluation context production above involves two language constructs, namely \texttt{spawn} and sequential composition. The desired non-determinism due to concurrency is captured by deliberate ambiguity in parsing evaluation contexts and, implicitly, in the splitting/plugging mechanism.

The remaining rule (\textsc{SmallStep-Spawn-Skip}) (resp. (\textsc{MSOS-Spawn-Skip})) in Section \textsection3.5.4 (resp. Section \textsection3.6.2) is turned into an equivalent rule here, and the structural identity stating the associativity of sequential composition and the syntactic extension of \texttt{spawn} to take statements (instead of blocks) are also still necessary:

\[
\texttt{spawn \{\}} \rightarrow \{\} \\
(s_1 \ s_2) \ s_3 \equiv s_1 \ (s_2 \ s_3)
\]

Local Variables

We make use of the procedure presented in Section \textsection3.5.5 for desugaring blocks with local variables into \texttt{let} constructs, to reduce the problem to only give semantics to \texttt{let}. Recall from Section \textsection3.5.5 that the semantics of \texttt{let} \( x=a \ \text{in} \ s \) in a state \( \sigma \) is to first evaluate arithmetic expression \( a \) in \( \sigma \) to some integer \( i \) and then evaluate statement \( s \) in state \( \sigma[i/x] \); after the evaluation of \( s \), the value of \( x \) is recovered to \( \sigma(x) \) (i.e., whatever it was before the execution of the block) but all the other state updates produced by the evaluation of \( s \) are kept. These suggest the following:

\[
\text{Context ::= ... | \texttt{let Id = Context in Stmt}} \\
\langle c, \sigma \rangle[\texttt{let} \ x=i \ \text{in} \ s] \rightarrow \langle c, \sigma[i/x] \rangle[s \ x = \sigma(x);]
\]
Notice that a solution similar to that in small-step SOS and MSOS (see rules (SmallStep-Let-Stmt) and (MSOS-Let-Stmt) in Sections 3.5.5 and 3.6.2 respectively) does not work here, because rules in reduction semantics with evaluation contexts are unconditional. In fact, a solution similar to the one we adopted above was already discussed in Section 3.5.6 in the context of small-step SOS, where we also explained that it works as shown because of a syntactic trick, namely because we allow assignment statements of the form “\(x = \perp\)” (in our case here when \(\sigma(x) = \perp\)), which have the effect to undefine \(x\) in \(\sigma\) (see Section 2.4.6). Section 3.5.6 also gives suggestions on how to avoid allowing \(\perp\) to be assigned to \(x\), if one does not like it. One suggestion was to rename the bound variable into a fresh one, this way relieving us from having to recover its value after the let:

\[
\langle c, \sigma \rangle[\text{let } x = i \text{ in } s] \rightarrow \langle c, \sigma[i/x'][s[x'/x]] \rangle \text{ if } x' \text{ is a fresh variable}
\]

This approach comes with several problems, though: it requires that we define and maintain a substitution operation (for \(s[x'/x]\)), we have to pay a complexity linear with the size of the let body each time a let statement is eliminated, and the state may grow indefinitely (since the let can be inside a loop).

Note also that, with the current reduction semantics with evaluation contexts of IMP, we cannot add the following evaluation context production

\[
\text{Context ::= ... | let Id = Int in Context}
\]

stating that once the binding expression becomes an integer then we can evaluate the let body. We cannot add it simply because we have to bind the variable to the integer before we evaluate the let body statement. An evaluation context production like above would result in evaluating the let body statement in the same state as before the let, which is obviously wrong. However, the existence of a let binder allows us to possibly rethink the overall reduction semantics of IMP, to make it more syntactic. Indeed, the let binders can be used in a nested manner and hereby allow us to syntactically mimic a state. For example, a statement of the form \(\text{let } x = 5 \text{ in let } y = 7 \text{ in } s\) can be thought of as the statement \(s\) being executed in a “state” where \(x\) is bound to 5 and where \(y\) is bound to 7. We can then drop the configurations of IMP completely and instead add the evaluation context above allowing reductions inside let body statements. The IMP rules that refer to configurations, namely those for lookup and assignment, need to change to work with the new “state”, and a new rule to eliminate unnecessary let statements needs to be added:

\[
\begin{align*}
\text{let } x = i & \text{ in } c[x] \rightarrow \text{let } x = i \text{ in } c[i] \\
\text{let } x = i & \text{ in } c[x = j] \rightarrow \text{let } x = j \text{ in } c[j] \\
\text{let } x = i & \text{ in } \{} \rightarrow \{}
\end{align*}
\]

The above works correctly only if one ensures that the evaluation contexts \(c\) do not contain other let \(x = \_\text{ in } \_\) evaluation context constructs, with the same \(x\) variable name as in the rules (the underscores can be any integer and context, respectively). One can do this by statically renaming the bound variables to have different names at parse-time, or by employing a substitution operation to do it dynamically. Note that the static renaming approach requires extra-care as the language is extended, particularly if new let statements can be generated dynamically by other language constructs. Another approach to ensure the correct application of the rules above, which is theoretically more complex and practically more expensive and harder to implement, is to add a side condition to the first two rules of the form “where \(c\) does not contain any evaluation context production instance of the form let \(x = \_\text{ in } \_\)”.

To conclude, the discussion above suggests that there are various ways to give a reduction semantics of let using evaluation contexts, none of them absolutely better than the others: some are simpler, others
are more syntactic but require special external support, others require non-modular changes to the existing language. The approaches above are by no means exhaustive. For example, we have not even discussed environment-store based approaches.

### Putting Them All Together

It is relatively easy to combine all the reduction semantics with evaluation contexts of all the features above, although not as modularly as it was for MSOS. Specifically, we have to do the following:

1. Apply all the changes that we applied when we added input/output to IMP above, namely: add input/output buffers to both configurations and configuration evaluation contexts; change the semantic rules involving configurations to work with the extended configurations, more precisely the rules for variable lookup, for variable assignment, and for programs.

2. Add all the evaluation contexts and the rules for the individual features above, making sure we change those of them using configurations or configuration evaluation contexts (i.e., almost all of them) to work with the new configurations including input/output buffers.

Unfortunately, the above is not giving us the desired language. Worse, it actually gives us a wrong language, namely one with a disastrous feature interaction. This problem has already been noted in Section 3.5.6 where we discussed the effect of a similar semantics to that of `let` above, but using small-step SOS instead of evaluation contexts. The problem here is that, if the body of a `let` statement contains a `spawn` statement, then the latter will be allowed, according to its semantics, to be executed in parallel with the statements following it. In our case, the assignment statement `x = σ(x)`; in the `let` semantics, originally intended to recover the value of `x`, can be now potentially executed before the spawned statement, resulting in a wrong behavior; in particular, the assignment can even “undefine” `x` in case `σ(x) = ⊥`, in which case the `spawn` statement can even get stuck.

As already indicated in Section 3.5.6 the correct way to eliminate the `let` construct is to rename the bound variable into a fresh variable visible only to `let`’s body statement, this way eliminating the need to recover the value of the bound variable to what it was before the `let`:

\[ \langle c, σ, ω_{in}, ω_{out} \rangle[let \ x = i \ in \ s] \rightarrow \langle c, σ[i/x'], ω_{in}, ω_{out}[s[x'/x]] \rangle \text{ if } x' \text{ is a fresh variable} \]

We have used configurations already extended with input/output buffers, as needed for IMP++. This solution completely brakes any (intended or unintended) relationship between the `let` construct and any other language constructs that may be used inside its body, although, as discussed in Section 3.5.6, the use of the substitution comes with a few (relatively acceptable) drawbacks.

### 3.7.3 Reduction Semantics with Evaluation Contexts in Rewrite Logic

In this section we show how to automatically and faithfully embed reduction semantics with evaluation contexts into rewrite logic. After discussing how to embed evaluation contexts into rewrite logic, we first give a straightforward embedding of reduction semantics, which is easy to prove correct but which does not take advantage of performance-improving techniques currently supported by rewrite engines, so consequently it is relatively inefficient when executed or formally analyzed. We then discuss simple optimizations which increase the performance of the resulting rewrite definitions an order of magnitude or more. We only consider evaluation contexts which can be defined by means of context-free grammars (CFGs). However, the CFG that we allow for defining evaluation contexts can be non-deterministic, in the sense that a term is allowed to split many different ways into a context and a redex (like the CFG in Figure 3.30).
sort:
Syntax  // includes all syntactic terms, in contextual representation context[redex] or not
subsorts:
N₁, N₂, . . . < Syntax  // N₁, N₂, . . ., are sorts whose terms can be regarded as context[redex]
operations:
[.] : Context × Syntax → Syntax  // constructor for terms in contextual representation
split : Syntax → Syntax  // puts syntactic terms into contextual representation
plug : Syntax → Syntax  // the dual of split
rules and equations:
split(Syn) → □[Syn]  // generic rule; it initiates the splitting process for the rules below
plug(□[Syn]) = Syn  // generic equation; it terminates the plugging process
// for each context production Context ::= π(N₁, . . . , Nₙ, Context) add the following:
split(π(T₁, . . . , Tₙ, T)) → π(T₁, . . . , Tₙ, C)[Syn]  if  split(T) → C[Syn]
plug(π(T₁, . . . , Tₙ, C)[Syn]) = π(T₁, . . . , Tₙ, plug(C[Syn]))

Figure 3.32: Embedding evaluation contexts into rewrite logic theory \( R_{\text{SEC}}^{\boxplus} \). The implicit split/plug mechanism is replaced by explicit rewrite logic sentences achieving the same task (the involved variables have the sorts Syn : Syntax, C : Context, T₁ : N₁, . . . , Tₙ : Nₙ, and T : N).

Faithful Embedding of Evaluation Contexts into Rewrite Logic

Our approach to embedding reduction semantics with evaluation contexts in rewrite logic builds on an embedding of evaluation contexts and their implicit splitting/plugging mechanism in rewrite logic. More precisely, each evaluation context production is associated with an equation (for plugging) and a conditional rewrite rule (for splitting). The conditional rewrite rules allow to non-deterministically split a term into a context and a redex. Moreover, when executing the resulting rewrite logic theory, the conditional rules allow for finding all splits of a term into a context and a redex, provided that the underlying rewrite engine has search capabilities (like Maude does).

Figure 3.32 shows a general and automatic procedure to generate a rewrite logic theory from any CFG defining evaluation contexts for some given language syntax. Recall that, for simplicity, in this section we assume only one Context syntactic category. What links the CFG of evaluation contexts to the CFG of the language to be given a semantics, which is also what makes our embedding into rewrite logic discussed here work, is the assumption that for any context production

\[ \text{Context} ::= \pi(N₁, \ldots, Nₙ, \text{Context}) \]

there are some syntactic categories \( N, N' \) (different or not) in the language CFG (possibly extended with configurations and semantic components as discussed above) such that \( \pi(t₁, \ldots, tₙ, t) \in N' \) for any \( t₁ \in N₁, \ldots, tₙ \in Nₙ, t \in N \). We here used a notation which needs to be explained. The actual production above is \( \text{Context} ::= \pi \), where \( \pi \) is a string of terminals and non-terminals, but we write \( \pi(N₁, \ldots, Nₙ, \text{Context}) \) instead of \( \pi \) to emphasize that \( N₁, \ldots, Nₙ, \text{Context} \) are all the non-terminals appearing in \( \pi \); we listed the Context last for simplicity. Also, by abuse of notation, we let \( \pi(t₁, \ldots, tₙ, t) \) denote the term obtained by substituting \( (t₁, \ldots, tₙ, t) \) for \( (N₁, \ldots, Nₙ, \text{Context}) \) in \( \pi \), respectively. So our assumption is that \( \pi(t₁, \ldots, tₙ, t) \) is well-formed under the syntax of the language whenever \( t₁, \ldots, tₙ, t \) are well-defined in the appropriate syntactic categories, that is, \( t₁ \in N₁, \ldots, tₙ \in Nₙ, t \in T \). This is indeed a very natural property of well-defined evaluation contexts for a given language, so natural that one may even ask how it can be otherwise (it is
easy to violate this property though, e.g., \( \text{Context} ::= \text{Context} <\text{AExp}; \). Without this property, our embedding of evaluation contexts in rewrite logic in Figure 3.32 would not be well-formed, because the left-hand-side terms of some of the conditional rule(s) for split would not be well-formed terms.

For simplicity, in Figure 3.32 we prefer to subsort all the syntactic categories whose terms are intended to be allowed contextual representations \( \text{context}[\text{redex}] \) under one top sort \( \text{Syntax} \). The implicit notation \( \text{context}[\text{term}] \) for contextual representations, as well as the implicitly assumed \textit{split} and \textit{plug} operations, are defined explicitly in the corresponding rewrite theory. The \textit{split} operation is only defined on terms over the original language syntax, while the \textit{plug} operation is defined only over terms in contextual representation. One generic rule and one generic equation are added: \( \text{split}(\text{Syn}) \rightarrow \Box[\text{Syn}] \) initiates the process of splitting a term into a contextual representation and \( \text{plug}(\Box[\text{Syn}]) = \text{Syn} \) terminates the process of plugging a term into a context. It is important that the first be a rewrite rule (because it can lead to non-determinism; this is explained below), while the second can safely be an equation.

Each evaluation context production translates into one equation and one conditional rewrite rule. The equation tells how terms are plugged into contexts formed with that production, while the conditional rule tells how that production can be used to split a term into a context and a redex. The equations defining plugging are straightforward: for each production in the original CFG of evaluation contexts, iteratively plug the subterm in the smaller context; when the hole is reached, replace it by the subterm via the generic equation. The conditional rules for splitting also look straightforward, but how and why they work is more subtle. For any context production, if the term to split matches the pattern of the production, then first split the subterm corresponding to the position of the subcontext and then use that contextual representation of the subterm to construct the contextual representation of the original term; at any moment, one has the option to stop splitting thanks to the generic rule \( \text{split}(\text{Syn}) \rightarrow \Box[\text{Syn}] \). For example, for the five \textit{Context} productions in the evaluation context CFG in the preamble of this section, namely

\[
\text{Context} ::= \text{Context} <\text{AExp} \\
| \quad \text{Int} <\text{Context} \\
| \quad \text{Id} = \text{Context}; \\
| \quad \text{Context Stmt} \\
| \quad \text{if} (\text{Context}) \text{Stmt else Stmt}
\]

the general procedure in the rewrite logic embedding of evaluation contexts in Figure 3.32 yields the following five rules and five equations (variable \( I_1 \) has sort \textit{Int}; \( X \) has sort \textit{Id}; \( A_1 \) and \( A_2 \) have sort \textit{AExp}; \( B \) has sort

---

9 An alternative, which does not involve subsorting, is to rename all syntactic categories into one, \textit{Syntax}. Our construction also works without subsorting and without collapsing of syntactic categories, but it is more technical, requires more operations, rules, and equations, and it is likely not worth the effort without a real motivation to use it in term rewrite settings without support for subsorting. We have made experiments with both approaches and found no penalty on performance when collapsing syntactic categories.
BExp; S₁ and S₂ have sort Stmt; C has sort Context; Syn has sort Syntax:

\[
\text{split}(A₁ \leq A₂) \rightarrow (C \leq A₂)[\text{Syn}] \quad \text{if} \quad \text{split}(A₁) \rightarrow C[\text{Syn}]
\]
\[
\text{plug}(C \leq A₂)[\text{Syn}] = \text{plug}(C[\text{Syn}]) \leq A₂
\]

\[
\text{split}(I₁ \leq A₂) \rightarrow (I₁ \leq C)[\text{Syn}] \quad \text{if} \quad \text{split}(A₂) \rightarrow C[\text{Syn}]
\]
\[
\text{plug}(I₁ \leq C)[\text{Syn}] = I₁ \leq \text{plug}(C[\text{Syn}])
\]

\[
\text{split}(X = A;) \rightarrow (X = C;)[\text{Syn}] \quad \text{if} \quad \text{split}(A) \rightarrow C[\text{Syn}]
\]
\[
\text{plug}((X = C;)[\text{Syn}]) = (X = \text{plug}(C[\text{Syn}]));
\]

\[
\text{split}(S₁ S₂) \rightarrow (C S₂)[\text{Syn}] \quad \text{if} \quad \text{split}(S₁) \rightarrow C[\text{Syn}]
\]
\[
\text{plug}(C S₂)[\text{Syn}] = \text{plug}(C[\text{Syn}]) S₂
\]

\[
\text{split}(\text{if} (B) S₁ \text{ else } S₂) \rightarrow (\text{if} (C) S₁ \text{ else } S₂)[\text{Syn}] \quad \text{if} \quad \text{split}(B) \rightarrow C[\text{Syn}]
\]
\[
\text{plug}(\text{if} (C) S₁ \text{ else } S₂)[\text{Syn}] = \text{if} (\text{plug}(C[\text{Syn}])) S₁ \text{ else } S₂
\]

The theorem above says that the process of splitting a term \( t \) into a context and a redex in reduction semantics with evaluation contexts, which can be non-deterministic, reduces to reachability in the corresponding rewrite logic theory of a contextual representation pattern \( c[r] \) of the original term marked for splitting, \( \text{split}(t) \). Rewrite engines such as Maude provide a search command that does precisely that. We will shortly see how Maude’s search command can find all splits of a term.

**Theorem 17. (Embedding splitting/plugging into rewrite logic)** Given an evaluation context CFG as discussed above, say as part of some reduction semantics with evaluation contexts definition RSEC, let \( \mathcal{R}^{\text{RSEC}}_\text{RSEC} \) be the rewrite logic theory associated to it as in Figure 3.32 Then the following are equivalent for any \( t, r \in \text{Syntax} \) and \( c \in \text{Context} \):

- \( t \) can be split as \( c[r] \) using the evaluation context CFG of RSEC;
- \( \mathcal{R}^{\text{RSEC}}_\text{RSEC} \vdash \text{split}(t) \rightarrow c[r]; \)
- \( \mathcal{R}^{\text{RSEC}}_\text{RSEC} \vdash \text{plug}(c[r]) = t. \)

The theorem above says that the process of splitting a term \( t \) into a context and a redex in reduction semantics with evaluation contexts, which can be non-deterministic, reduces to reachability in the corresponding rewrite logic theory of a contextual representation pattern \( c[r] \) of the original term marked for splitting, \( \text{split}(t) \). Rewrite engines such as Maude provide a search command that does precisely that. We will shortly see how Maude’s search command can find all splits of a term.

**Faithful Embedding of RSEC in Rewriting Logic**

In this section we discuss three faithful rewrite logic embeddings of reduction semantics with evaluation contexts. The first two assume that the embedded reduction semantics has no characteristic rule, in that all reductions take place at the top of the original term to reduce (e.g., a configuration in the case of our IMP language); this is not a limitation because, as already discussed, the characteristic rule can be regarded as syntactic sugar anyway, its role being to allow one to write reduction semantics definitions more compactly and elegantly. The first embedding is the simplest and easiest to prove correct, but it is the heaviest in notation

245
of rules:

\[
\begin{align*}
&\text{// for each reduction semantics rule } l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow r(c'_1[r_1], \ldots, c'_m[r_m]) \\
&\text{// add the following conditional semantic rewrite rule:}
\end{align*}
\]

\[
\circ \bar{T}(I_1, \ldots, T_n) \rightarrow \bar{r}(\overline{\text{plug}(c'_1[I_1]), \ldots, \overline{\text{plug}(c'_m[I_m])}) \quad \text{if \hspace{1mm} split}(I_1) \rightarrow \overline{I_1} \land \ldots \land \text{split}(T_n) \rightarrow \overline{T_n}
\]

Figure 3.33: First embedding of RSEC into rewrite logic (\(\text{RSEC} \rightarrow \mathcal{R}_\text{RSEC}\)).

and the resulting rewrite theories tend to be inefficient when executed because most of the left-hand-side terms of rules end up being identical, thus making the task of matching and selecting a rule to apply rather complex for rewrite engines. The second embedding results in rewrite rules whose left-hand-sides are mostly distinct, thus taking advantage of current strengths of rewrite engines to index terms so that rules to be applied can be searched for quickly. Our third embedding is as close to the original reduction semantics in form and shape as one can hope it to be in a rewriting setting; in particular, it also defines a characteristic rule, which can be used to write a more compact semantics. The third embedding yields rewrite theories which are as efficient as those produced by the second embedding. The reason we did not define directly the third embedding is because we believe that the transition from the first to the second and then to the third is instructive.

Since reduction semantics with evaluation contexts is an inherently small-step semantical approach, we use the same mechanism to control the rewriting as for small-step SOS (Section 3.3) and MSOS (Section 3.6). This mechanism was discussed in detail in Section 3.3.3. It essentially consists of: (1) tagging each left-hand-side term appearing in a rule transition with a \(\circ\), to capture the desired notion of a one-step reduction of that term; and (2) tagging with a \(\star\) the terms to be multi-step (zero, one or more steps) reduced, where \(\star\) can be easily defined with a conditional rule as the transitive and reflexive closure of \(\circ\) (see Section 3.3.3).

Figure 3.33 shows our first embedding of reduction semantics with evaluation contexts into rewrite logic, which assumes that the characteristic rule, if any, has already been desugared. Each reduction semantics rule translates into one conditional rewrite rule. We allow the reduction rules to have in their left-hand-side and right-hand-side terms an arbitrary number of subterms that are in contextual representation. For example, if the left-hand-side \(l\) of a reduction rule has \(n\) such subterms, say \(c_1[l_1], \ldots, c_n[l_n]\), then we write it \(l(c_1[l_1], \ldots, c_n[l_n])\) (this is similar with our previous notation \(\pi(N_1), \ldots, N_n, N\) in the section above on embedding of evaluation contexts into rewrite logic, except that we now single out all the subterms in contextual representation instead of all the non-terminals). In particular, a rule \(l \rightarrow r\) in which \(l\) and \(r\) contain no subterms in contextual representation (like the last rule in Figure 3.31) is translated exactly like in small-step SOS, that is, into \(\bar{l} \rightarrow \bar{r}\). Also, note that we allow evaluation contexts to have any pattern (since we overline them, like any other terms); we do not restrict them to only be context variables. Consider, for example, the six reduction rules discussed in the preamble of Section 3.7, which after the desugaring of the characteristic rule are as follows:

\[
\begin{align*}
&c[i_1 \leq i_2] \rightarrow c[i_1 \leq_{\text{int}} i_2] \\
&c[\{\}\_2] \rightarrow c[s_2] \\
&c[\text{if (true)} s_1 \text{ else } s_2] \rightarrow c[s_1] \\
&c[\text{if (false)} s_1 \text{ else } s_2] \rightarrow c[s_2] \\
&\langle c, \sigma \rangle[x] \rightarrow \langle c, \sigma \rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot \\
&\langle c, \sigma \rangle[x = i] \rightarrow \langle c, \sigma[i/x]\rangle[\{\}] \quad \text{if } \sigma(x) \neq \bot
\end{align*}
\]

Since all these rules have left-hand-side terms already in contextual representation, their corresponding \(l\) in
Theorem 18. (First faithful embedding of reduction semantics into rewrite logic) Let RSEC be any reduction semantics with evaluation contexts definition and let $R_{\text{RSEC}}$ be the rewrite logic theory associated to RSEC using the embedding procedures in Figures 3.32 and 3.33. Then

1. (step-for-step correspondence) RSEC $\vdash t \rightarrow t'$ using a reduction semantics with evaluation contexts rule iff $R_{\text{RSEC}} \vdash \circ \tilde{t} \rightarrow \tilde{t'}$ using the corresponding conditional rewrite rule obtained like in Figure 3.33; moreover, the reduction rule and the corresponding rewrite rule apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 17);

2. (computational correspondence) RSEC $\vdash t \rightarrow^* t'$ iff $R_{\text{RSEC}} \vdash \star \tilde{t} \rightarrow \star \tilde{t'}$.

The first item in Theorem 18 says that the resulting rewriting logic theory captures faithfully the small-step reduction relation of the original reduction semantics with evaluation contexts definition. The faithfulness
of this embedding (i.e., there is precisely one top-level application of a rewrite rule that corresponds to an application of a reduction semantics rule), comes from the fact that the consistent use of the $\circ$ tag inhibits any other application of any other rule on the tagged term. Therefore, like in small-step SOS and MSOS, a small-step in a reduction semantics definition also reduces to reachability analysis in the corresponding rewrite theory; one can also use the search capability of a system like Maude to find all the next terms that a given term evaluates to (Maude provides the capability to search for the first $n$ terms that match a given pattern using up to $m$ rule applications, where $n$ and $m$ are user-provided parameters).

The step-for-step correspondence above is stronger (and better) than the strong bisimilarity of the two definitions; for example, if a reduction semantics rule in RSEC can be applied in two different ways on a term to reduce, then its corresponding rewrite rule in $R_{1RSEC}$ can also be applied in two different ways on the tagged term. The second item in Theorem 18 says that the resulting rewrite theory can be used to perform any computation possible in the original RSEC, and vice versa (the step-for-step correspondence is guaranteed in combination with the first item). Therefore, there is absolutely no difference between computations using RSEC and computations using $R_{1RSEC}$, except for irrelevant syntactic conventions/notations. This strong correspondence between reductions in RSEC and rewrites in $R_{1RSEC}$ tells that $R_{1RSEC}$ is precisely RSEC, not an encoding of it. In other words, RSEC can be faithfully regarded as a methodological fragment of rewrite logic, same like big-step SOS, small-step SOS, and MSOS.

The discussion above implies that, from a theoretical perspective, the rewrite logic embedding of reduction semantics in Figure 3.33 is as good as one can hope. However, its simplicity comes at a price in performance, which unfortunately tends to be at its worst precisely in the most common cases. Consider, for example, the six rewrite rules used before Theorem 18 to exemplify the embedding in Figure 3.33 (consider the variant for lookup and assignment rules where the contextual representation in the left-hand-side appears at the top—first variant). They all have the form:

$$\circ Cfg \rightarrow \ldots \text{ if } split(Cfg) \rightarrow \ldots$$

In fact, as seen in Figure 3.38 all the rewrite rules in the rewrite logic theory corresponding to the RSEC of IMP have the same form. The reason the left-hand-side terms of these rewrite rules are the same and lack any structure is because the contextual representations in the left-hand-side terms of the RSEC rules appear at the top, with no structure above them, which is the most common type of RSEC rule encountered.

To apply a conditional rewrite rule, a rewrite engine first matches the left-hand-side and then performs the (exhaustive) search in the condition. In other words, the structure of the left-hand-side acts as a cheap guard for the expensive search. Unfortunately, since the left-hand-side of the conditional rewrite rules above has no structure, it will always match. That means that the searches in the conditions of all the rewrite rules will be, in the worst case, executed one after another until a split is eventually found (if any). If one thinks in terms of implementing RSEC in general, then this is what a naive implementation would do. If one thinks in terms of executing term rewrite systems, then this fails to take advantage of some important performance-increasing advances in term rewriting, such as indexing [72,73,2]. In short, indexing techniques use the structure of the left-hand-sides to augment the term structure with information about which rule can potentially be applied at which places. This information is dynamically updated, as the term is rewritten. If the rules’ left-hand-sides do not significantly overlap, it is generally assumed that it takes constant time to find a matching rewrite rule. This is similar in spirit to hashing, where the access time into a hash table is generally assumed to take constant time when there are no or few key collisions. Thinking intuitively in terms of hashing, from an indexing perspective a rewrite system with rules having the same left-hand-sides is as bad as a hash table in which all accesses are collisions.

Ideally, in an efficient implementation of RSEC one would like to adapt/modify indexing techniques,
rules:

// for each term \( l \) that appears as left-hand-side of a reduction rule
// \( l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow \ldots \) with \( n > 0 \), add the following
// conditional rewrite rule (there could be one \( l \) for many reduction rules):
\[
\circ \tilde{l}(T_1, \ldots, T_n) \rightarrow T \text{ if } \circ \tilde{l}(\text{split}(T_1), \ldots, \text{split}(T_n)) \rightarrow T
\]

// for each reduction semantics rule \( l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow r(c'_1[r_1], \ldots, c'_n[r_n]) \)
// add the following (unconditional) semantic rewrite rule:
\[
\circ \tilde{l}(\overline{c}_1[l_1], \ldots, \overline{c}_n[l_n]) \rightarrow \overline{r}(\text{plug}(c'_1[r_1]), \ldots, \text{plug}(c'_n[r_n]))
\]

Figure 3.34: Second embedding of RSEC into rewrite logic (RSEC \( \rightsquigarrow R^R_{RSEC} \)).

which currently work for context-insensitive term rewriting, or to invent new techniques serving the same purpose. This seems highly non-trivial and tedious, though. An alternative is to device embedding transformations of RSEC into rewrite logic that take better or full advantage of existing, context-insensitive indexing. Without context-sensitive indexing or other bookkeeping mechanisms hardwired in the reduction engine, due to the inherent non-determinism in parsing/splitting syntax into contextual representations, in the worst case one needs to search the entire term to find a legal position where a reduction can take place. While there does not seem that we can do much to avoid such an exhaustive search in the worst case, note that our first embedding in Figure 3.33 initiates such a search in the condition of every rewrite rule: since in practice many/most of the rewrite rules generated by the procedure in Figure 3.33 end up having the same left-hand-side, the expensive search for appropriate splittings is potentially invoked many times. What we’d like to achieve is: (1) activate the expensive search for splitting only once; and (2) for each found split, quickly test which rule applies and apply it. Such a quick test as desired in (2) can be achieved for free on existing rewrite systems that use indexing, such as Maude, if one slightly modifies the embedding translation of RSEC into rewrite logic as shown in Figure 3.34.

The main idea is to keep the structure of the left-hand-side of the RSEC rules in the left-hand-side of the corresponding rewrite rules. This structure is crucial for indexing. To allow it, one needs to do the necessary splitting as a separate step. The first type of rewrite rules in Figure 3.34, one per term appearing as a left-hand-side in any of the conditional rules generated following the first embedding in Figure 3.33, enables the splitting process on the corresponding contextual representations in the left-hand-side of the original RSEC rule. We only define such rules for left-hand-side terms having at least one subterm in contextual representation, because if the left-hand-side \( l \) has no such terms then the rule would be \( \circ \tilde{l} \rightarrow T \) if \( \circ \tilde{l} \rightarrow T \), which is useless and does not terminate.

The second type of rules in Figure 3.34, one per RSEC rule, have almost the same left-hand-sides as the original RSEC rules; the only difference is the algebraic notation (as reflected by the overlining). Their right-hand-sides plug the context representations, so that they always yield terms which are well-formed over the original syntax (possibly extended with auxiliary syntax for semantics components—configurations, states, etc.). Consider, for example, the six RSEC rules discussed in the preamble of Section 3.7, whose translation into rewrite rules following our first embedding in Figure 3.33 was discussed right above Theorem 18. Let us first consider the variant for lookup and assignment rules where the contextual representation in the left-hand side appears at the top. Since in all these rules the contextual representation appears at the top of their
With this, the six rewrite rules of the second type in Figure 3.34 corresponding to the six RSEC rules under discussion are the following:

\[ \circ \text{C} \circ \rightarrow \circ \text{C} \circ \text{f} \circ \circ \text{split}(\text{C} \circ) \rightarrow \circ \text{C} \circ \text{f} \circ \]

With this, the six rewrite rules of the second type in Figure 3.34 corresponding to the six RSEC rules under discussion are the following:

\[
\begin{align*}
\circ C[I_1 := I_2] & \rightarrow \text{plug}(C[I_1 \leq_{\text{int}} I_2]) \\
\circ C[\text{if} (\text{true}) \ S_1 \ \text{else} \ S_2] & \rightarrow \text{plug}(C[S_1]) \\
\circ C[\text{if} (\text{false}) \ S_1 \ \text{else} \ S_2] & \rightarrow \text{plug}(C[S_2]) \\
\circ \langle C, \sigma \rangle[X] & \rightarrow \text{plug}(\langle C, \sigma \rangle(\sigma(X))) \quad \text{if} \ \sigma(X) \neq \bot \\
\circ \langle C, \sigma \rangle[X = I; ] & \rightarrow \text{plug}(\langle C, \sigma[I/X]\rangle[I]) \quad \text{if} \ \sigma(X) \neq \bot
\end{align*}
\]

If one prefers the second variant for the reduction rules of lookup and assignment, namely

\[
\begin{align*}
\langle \text{c}[x], \sigma \rangle & \rightarrow \langle \text{c}[\sigma(x)], \sigma \rangle \quad \text{if} \ \sigma(x) \neq \bot \\
\langle \text{c}[x = \text{t}; ], \sigma \rangle & \rightarrow \langle \text{c}[][], \sigma[\text{t}/\text{x}] \rangle \quad \text{if} \ \sigma(x) \neq \bot
\end{align*}
\]

then, since the left-hand-side of these rules is a pattern of the form \( \langle \text{Stmt}, \sigma \rangle \) which in algebraic form (over-lined) becomes a term of the form \( \langle S, \sigma \rangle \), we need to add one more rewrite rule of the first type in Figure 3.34, namely

\[ \circ \langle S, \sigma \rangle \rightarrow \text{C} \circ \circ \text{ff} \circ \circ \text{split}(S), \sigma \circ \rightarrow \circ \text{C} \circ \text{ff} \circ \]

and to replace the rewrite rules for lookup and assignment above with the following two rules:

\[
\begin{align*}
\circ \langle C[X], \sigma \rangle & \rightarrow \langle \text{plug}(C[\sigma(X)]), \sigma \rangle \quad \text{if} \ \sigma(X) \neq \bot \\
\circ \langle C[X = I; ], \sigma \rangle & \rightarrow \langle \text{plug}(C[][]), \sigma[I/X]\rangle \quad \text{if} \ \sigma(X) \neq \bot
\end{align*}
\]

**Theorem 19.** (Second faithful embedding of reduction semantics in rewrite logic) Let RSEC be any reduction semantics with evaluation contexts definition and let \( \mathcal{R}_{\text{RSEC}} \) be the rewrite logic theory associated to RSEC using the embedding procedures in Figures 3.32 and 3.34. Then

1. **(step-for-step correspondence)** RSEC \( \vdash t \rightarrow t' \) using a reduction semantics with evaluation contexts rule if \( \mathcal{R}_{\text{RSEC}} \vdash \circ t \rightarrow^1 t \) using the corresponding rewrite rules obtained like in Figure 3.34 (first a conditional rule of the first type whose left-hand-side matches \( t \), then a rule of the second type which solves, in one rewrite step, the condition of the first rule): moreover, the reduction rule and the corresponding rewrite rules apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 17);

2. **(computational correspondence)** RSEC \( \vdash t \rightarrow^* t' \) iff \( \mathcal{R}_{\text{RSEC}} \vdash \star t \rightarrow^* \star t' \).

Theorem 19 tells us that we can use our second rewriting logic embedding transformation in Figure 3.34 to seamlessly execute RSEC definitions on context-insensitive rewrite engines, such as Maude. This was also the case for our first embedding (Figure 3.33 and its corresponding Theorem 18). However, as explained above, in our second embedding the left-hand-side terms of the rewrite rules corresponding to the actual reduction semantics rules (the second type of rule in Figure 3.34) preserve the structure of the left-hand-side
rules:
// for each term \( l \) that appears as the left-hand-side of a reduction rule
// \( l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow \ldots \), add the following conditional
// rewrite rule (there could be one \( l \) for many reduction rules):
\[
\circ \bar{l}(T_1, \ldots, T_n) \rightarrow T \text{ if } \text{plug}(\circ \bar{l}(\text{split}(T_1), \ldots, \text{split}(T_n))) \rightarrow T
\]
// for each non-identity term \( r \) appearing as right-hand-side in a reduction rule
// \( \ldots \rightarrow r(c_1[r_1], \ldots, c_n[r_n]) \), add the following equation
// (there could be one \( r \) for many reduction rules):
\[
\text{plug}(\bar{\text{Syn}}_1, \ldots, \bar{\text{Syn}}_n) = \bar{r}(\text{plug}(\bar{\text{Syn}}_1), \ldots, \text{plug}(\bar{\text{Syn}}_n))
\]
// for each reduction semantics rule \( l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow r(c'_1[r_1], \ldots, c'_n[r'_n]) \)
// add the following semantic rewrite rule:
\[
\circ \bar{l}(\bar{c}_1[\bar{l}_1], \ldots, \bar{c}_n[\bar{l}_n]) \rightarrow \bar{r}(\bar{c}_1'[\bar{r}_1], \ldots, \bar{c}_n'[\bar{r}_n'])
\]

Figure 3.35: Third embedding of RSEC in rewrite logic (RSEC \( \sim RSEC_{RSEC} \)).

terms of the original corresponding reduction rules. This important fact has two benefits. On the one hand, the underlying rewrite engines can use that structure to enhance the efficiency of rewriting by means of indexing, as already discussed above. On the other hand, the resulting rewrite rules resemble the original reduction rules, so the language designer who wants to use our embedding feels more comfortable. Indeed, since the algebraic representation of terms (the overline) should not change the way they are perceived by a user, the only difference between the left-hand-side of the original reduction rule and the left-hand-side of the resulting rewrite rule is the \( \circ \) symbol: \( l(c_1[l_1], \ldots, c_n[l_n]) \) versus \( \circ \bar{l}(\bar{c}_1[\bar{l}_1], \ldots, \bar{c}_n[\bar{l}_n]) \), e.g., \( \langle c[x = i];, \sigma \rangle \) versus \( \langle C[X = I];, \sigma \rangle \), where \( c, \sigma, x, i \) are reduction rule parameters while \( C, \sigma, X, I \) are corresponding variables of appropriate sorts.

Even though the representational distance between the left-hand-side terms in the original reduction rules and the left-hand-side terms in the resulting rewrite rules is minimal (one cannot eliminate the \( \circ \), as extensively discussed in Section 3.3.3), unfortunately, the same does not hold true for the right-hand-side terms. Indeed, a right-hand-side \( r(c_1[r_1], \ldots, c'_n[r'_n]) \) of a reduction rule becomes the right-hand-side \( \bar{r}(\text{plug}(\bar{c}_1[\bar{r}_1]), \ldots, \text{plug}(\bar{c}_n[\bar{r}_n])) \) of its corresponding rewrite rule, e.g., \( \langle c[\{} \rangle, \sigma \rangle \) becomes \( \langle \text{plug}(C[\{}), \sigma \rangle \).

Figure 3.35 shows our third and final embedding of RSEC in rewrite logic, which has the advantage that it completely isolates the uses of \( \text{split} / \text{plug} \) from the semantic rewrite rules. Indeed, the rewrite rule associated to a reduction rule has the same left-hand-side as in the second embedding, but now the right-hand-side is actually the algebraic variant of the right-hand-side of the original reduction rule. This is possible because of two simple adjustments of the second embedding:

1. To avoid having to explicitly use the \( \text{plug} \) operation in the semantic rewrite rules, we replace the first type of conditional rewrite rules in the second embedding, namely
\[
\circ \bar{l}(T_1, \ldots, T_n) \rightarrow T \text{ if } \circ \bar{l}(\text{split}(T_1), \ldots, \text{split}(T_n)) \rightarrow T,
\]
with slightly modified conditional rewrite rules of the form
\[ Cfg \rightarrow Cfg' \text{ if } \text{plug}(\circ\text{split}(Cfg)) \rightarrow Cfg' \]
\[ \langle S, \sigma \rangle \rightarrow Cfg' \text{ if } \text{plug}(\circ\text{split}(S, \sigma)) \rightarrow Cfg' \]

We also have two right-hand-side patterns in these reduction rules, namely Configuration and \(\langle \text{Stmt}, \sigma \rangle\), so we have the following two rules of the first type in Figure 3.35:

\[ \circ C[I_n \leq I_2] \rightarrow C[I_n \leq_{\text{int}} I_2] \]
\[ \circ C[\text{true} S_2] \rightarrow C[S_2] \]
\[ \circ C[\text{false} S_2] \rightarrow C[S_2] \]
\[ \circ \langle C[X = I_1], \sigma \rangle \rightarrow \langle C[X], \sigma(X) \rangle \text{ if } \sigma(X) \neq \bot \]
\[ \circ \langle C[X = I_1 \text{ or } X], \sigma \rangle \rightarrow \langle C[X], \sigma(X) \rangle \text{ if } \sigma(X) \neq \bot \]

We now give the six rewrite rules corresponding to the six reduction rules in discussion:

\[ \circ C[Syn] \rightarrow C[Syn'] \text{ if } C \neq \square \land \circ \text{Syn} \rightarrow \text{Syn}' \]

Note that we first check whether the context is proper in the condition of the characteristic rewrite rule above, and then we initiate a (small-step) reduction of the redex (by tagging it with the symbol \(\circ\)). The condition
is well-defined in rewrite logic because, as explained in Figure 3.32, we subsorted all the syntactic sorts together with the configuration under the top sort Syntax, so all these sorts belong to the same kind (see Section 2.5), which means that the operation ◦ can apply to any of them, including to Syntax, despite the fact that it was declared to take a Configuration to an ExtendedConfiguration (like in Section 3.3.3). With this characteristic rewrite rule, we can now restate the six rewrite rules corresponding to the six reduction rules above as follows:

\[
\begin{align*}
\circ I_1 & \leq I_2 \rightarrow I_1 \leq_{ls} I_2 \\
\circ \{\} & S_2 \rightarrow S_2 \\
\circ \text{if (true)} S_1 \text{else } S_2 & \rightarrow S_1 \\
\circ \text{if (false)} S_1 \text{else } S_2 & \rightarrow S_2 \\
\circ \langle C[X], \sigma \rangle & \rightarrow \langle C[\sigma(X)], \sigma \rangle \text{ if } \sigma(X) \neq \perp \\
\circ \langle C[X = I_1], \sigma \rangle & \rightarrow \langle C[], \sigma[I/X] \rangle \text{ if } \sigma(X) \neq \perp
\end{align*}
\]

Note, again, that ◦ is applied on arguments of various sorts in the same kind with Configuration.

The need for ◦ in the left-hand-side terms of rules like above is now even more imperative than before. In addition to all the reasons discussed so far, there are additional reasons now for which the dropping of ◦ would depart us from the intended faithful capturing of reduction semantics in rewrite logic. Indeed, if we drop ◦ then there is nothing to stop the applications of rewrite rules at any places in the term to rewrite, potentially including places which are not allowed to be evaluated yet, such as, for example, in the branches of a conditional. Moreover, such applications of rules could happen concurrently, which is strictly disallowed by reduction semantics with or without evaluation contexts. The role of ◦ is precisely to inhibit the otherwise unrestricted potential to apply rewrite rules everywhere and concurrently: rules are now applied sequentially and only at the top of the original term, exactly like in reduction semantics.

**Theorem 20. (Third faithful embedding of reduction semantics into rewrite logic)** Let RSEC be any reduction semantics with evaluation contexts definition (with or without a characteristic reduction rule) and let \( R_{RSEC} \) be the rewrite logic theory associated to RSEC using the embedding procedures in Figures 3.32 and 3.35 (plus the characteristic rewrite rule above in case RSEC comes with a characteristic reduction rule). Then

1. (step-for-step correspondence) RSEC ⊢ t → t' using a reduction semantics with evaluation contexts rule iff \( R_{RSEC} \) ⊢ ◦ t → ◦ t';

2. (computational correspondence) RSEC ⊢ t →* t' iff \( R_{RSEC} \) ⊢ * t → * t'.

We can therefore safely conclude that RSEC has been captured as a methodological fragment of rewrite logic. The faithful embeddings of reduction semantics into rewrite logic above can be used in at least two different ways. On the one hand, they can be used as compilation steps transforming a context-sensitive reduction system into an equivalent context-insensitive rewrite system, which can be further executed/compiled/analyzed using conventional rewrite techniques and existing rewrite engines. On the other hand, the embeddings above are so simple, that one can simply use them manually and thus “think reduction semantics” in rewrite logic.

**Reduction Semantics with Evaluation Contexts of IMP in Rewrite Logic**

We here discuss the complete reduction semantics with evaluation contexts definition of IMP in rewrite logic, obtained by applying the faithful embedding techniques discussed above to the reduction semantics definition of IMP in Figure 3.31 in Section 3.7.1. We start by defining the needed configurations, then we give all the
Figure 3.36: Configurations and infrastructure for the rewrite logic embedding of RSEC(IMP).

Figure 3.36 gives an algebraic definition of IMP configurations as needed for reduction semantics with evaluation contexts, together with the additional infrastructure needed to represent the one-step and multi-step transition relations. Everything defined in Figure 3.36 has already been discussed in the context of small-step SOS (see Figures 3.13 and 3.17 in Section 3.3.3). Note, however, that we only defined a subset of the configurations needed for small-step SOS, more precisely only the top-level configurations (ones holding a program and ones holding a statement and a state). The intermediate configurations holding expressions and a state in small-step SOS are not needed here because reduction semantics with evaluation contexts does not need to explicitly decompose bigger reduction tasks into smaller ones until a redex is eventually found, like small-step SOS does; instead, the redex is found atomically by splitting the top level configuration into a context and the redex.

Figure 3.37 shows the rewrite logic theory $R_{RSEC(IMP)}$ associated to the evaluation contexts of IMP in RSEC(IMP) (Figure 3.30) following the procedure described in Section 3.7.3 and summarized in Figure 3.32. Recall that all language syntactic categories and configurations are sunk into a top sort `Syntax`, and that one rule for splitting and one equation for plugging are generated for each context production. In general, the embedding of evaluation contexts tends to be the largest and the most boring portion of the rewrite logic embedding of a reduction semantics language definition. However, fortunately, this can be generated fully automatically. An implementation of the rewrite logic embedding techniques discussed in this section may even completely hide this portion from the user. We show it in Figure 3.37 only for the sake of completeness.

Figure 3.38 shows the rewrite logic theory $R_{RSEC(IMP)}$ corresponding to the rules in the reduction semantics with evaluation contexts of IMP in Section 3.7.1 following our first embedding transformation depicted in Figure 3.33. Like before, we used the rewrite logic convention that variables start with upper-case letters; if they are Greek letters, then we use a similar but larger symbol (e.g., $\sigma$ instead of $\sigma$ for variables of sort `State`). These rules are added, of course, to those corresponding to evaluation contexts in Figure 3.37 (which are common to all three embeddings). Note that there is precisely one conditional rewrite rule in Figure 3.38 corresponding to each reduction semantics rule of IMP in Figure 3.31. Also, note that if a rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3. For example, the last reduction rule in Figure 3.31 results in the last rewrite rule in Figure 3.38 which is identical to the last rewrite rule
sorts:
  Syntax, Context
subsorts:
  AExp, BExp, Stmt, Configuration < Syntax
operations:
\[ \square : \rightarrow Context \quad \\ \\ \ \langle \_ \_ \rangle : Context \times State \rightarrow Context \]
\[ split : Syntax \rightarrow Syntax \quad plug : Syntax \rightarrow Syntax \]
\[ (_\_+) : Context \times AExp \rightarrow Context \quad (_\_+) : AExp \times Context \rightarrow Context \]
\[ (_\_-) : Context \times AExp \rightarrow Context \quad (_\_-) : AExp \times Context \rightarrow Context \]
\[ (_\_<=) : Context \times AExp \rightarrow Context \quad (_\_<=) : Int \times Context \rightarrow Context \]
\[ !_: Context \rightarrow Context \quad \_\_&\_\_ : Context \times BExp \rightarrow Context \]
\[ _\_=_: _\_ : Id \times Context \rightarrow Context \quad _\_ _\_ : Context \times Stmt \rightarrow Context \]
\[ if(_) else __: Context \times Stmt \times Stmt \rightarrow Context \]

deduction semantics mechanism is replaced by explicit rewrite logic sentences.

Figure 3.37: $R_{\text{RSEC(IMP)}}^\circ$; Rewrite logic embedding of IMP evaluation contexts. The implicit split/plug reduction semantics mechanism is replaced by explicit rewrite logic sentences.
Like in the second embedding, there is precisely one unconditional rewrite rule corresponding to each RSEC in Figure 3.35). These rules are also added to those corresponding to evaluation contexts in Figure 3.37 and, language designers as an inconvenience.

generated relatively automatically, the remaining rewrite rules that correspond to the reduction rules still

◦ \( \text{Cfg} \rightarrow \text{plug}(⟨C, \sigma⟩[\sigma(X)]) \) if split(Cfg) \( \rightarrow \langle C, \sigma⟩[X] \land \sigma(X) \neq \bot \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨I_1 + I_2⟩) \) if split(Cfg) \( \rightarrow C[I_1 + I_2] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨I_1 / I_2⟩) \) if split(Cfg) \( \rightarrow C[I_1 / I_2] \land I_2 \neq 0 \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨I_1 \leq I_2⟩) \) if split(Cfg) \( \rightarrow C[I_1 \leq I_2] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨C[false⟩) \) if split(Cfg) \( \rightarrow C[! true] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨C[true⟩) \) if split(Cfg) \( \rightarrow C[! false] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨C[B_2⟩) \) if split(Cfg) \( \rightarrow C[true \& B_2] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨C[false⟩) \) if split(Cfg) \( \rightarrow C[false \& B_2] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(⟨C[I/X⟩)([I]) \) if split(Cfg) \( \rightarrow \langle C, \sigma⟩[X = I ;] \land \sigma(X) \neq \bot \)
◦ \( \text{Cfg} \rightarrow \text{plug}(C[I_2]) \) if split(Cfg) \( \rightarrow C[I_2] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(C[S_1]) \) if split(Cfg) \( \rightarrow C[if (true) S_1 else S_2] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(C[S_2]) \) if split(Cfg) \( \rightarrow C[if (false) S_1 else S_2] \)
◦ \( \text{Cfg} \rightarrow \text{plug}(C[if (B) S while (B) S ] else []) \) if split(Cfg) \( \rightarrow C[while (B) S] \)
◦ \( \langle \text{int X}!; S \rangle \rightarrow \langle S, (X! \rightarrow 0) \rangle \)

Figure 3.38: \( R_{RSEC(IMP)}^{\text{RSEC(IMP)}} \) — rewrite logic theory corresponding to the first embedding of the reduction semantics with evaluation contexts of IMP.

corresponding to the small-step SOS of IMP in Figure 3.18. The rules that make use of evaluation contexts perform explicit splitting (in the left-hand-side of the condition) and plugging (in the right-hand-side of the conclusion) operations. As already discussed but worth reemphasizing, the main drawbacks of this type of rewrite logic embedding are: (1) the expensive, non-deterministic search involving splitting of the original term is performed for any rule, and (2) it does not take advantage of one of the major optimizations of rewrite engines, indexing, which allows for quick detection of matching rules based on the structure of their left-hand-side terms.

Figure 3.39 shows the rewrite logic theory \( R_{RSEC(IMP)}^{\text{RSEC(IMP)}} \) that follows our second embedding transformation depicted in Figure 3.34. These rules are also added to those corresponding to evaluation contexts in Figure 3.37. Note that now there is precisely one unconditional rewrite rule corresponding to each reduction semantics rule of IMP in Figure 3.31 and that, unlike in the first embedding in Figure 3.38, the left-hand-side of each rule preserves the exact structure of the left-hand-side of the original reduction rule (after desugaring of the characteristic rule), so this embedding takes advantage of indexing optimizations in rewrite engines. Like in the first embedding, if a reduction rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3 (e.g., the last rule). Unlike in the first embedding, we also need to add a generic conditional rule, the first one in Figure 3.39 which initiates the splitting. We need only one rule of this type because all the left-hand-side terms of reduction rules of IMP in Figure 3.31 that contain a subterm in contextual representation contain that term at the top. As already discussed, if one preferred to write, e.g., the lookup RSEC rule as \( \langle C[x], \sigma \rangle \rightarrow \langle C[σ(x)], \sigma \rangle \) if \( σ(x) \neq \bot \), then one would need an additional generic rule, namely \( \langle S, \sigma \rangle \rightarrow Cfg' \) if \( \langle \text{split}(S), \sigma \rangle \rightarrow Cfg' \). While these generic rules take care of splitting and can be generated relatively automatically, the remaining rewrite rules that correspond to the reduction rules still make explicit use of the internal (to the embedding) plug operation, which can arguably be perceived by language designers as an inconvenience.

Figure 3.40 shows the rewrite logic theory \( R_{RSEC(IMP)}^{\text{RSEC(IMP)}} \) obtained by applying our third embedding (shown in Figure 3.35). These rules are also added to those corresponding to evaluation contexts in Figure 3.37 and, like in the second embedding, there is precisely one unconditional rewrite rule corresponding to each RSEC
\( \circ \text{Cfg} \rightarrow \text{Cfg}' \) if \( \circ \text{plug(split(Cfg))} \rightarrow \text{Cfg}' \)

\( \circ (C, \sigma)[X] \rightarrow \text{plug}(\langle C, \sigma \rangle[[\sigma(X)]] \) if \( \sigma(X) \neq \perp \)

\( \circ C[I_1 + I_2] \rightarrow \text{plug}(C[I_1 +_{\text{int}} I_2]) \)

\( \circ C[I_1 / I_2] \rightarrow \text{plug}(C[I_1 /_{\text{int}} I_2]) \) if \( I_2 \neq 0 \)

\( \circ C[I_1 <= I_2] \rightarrow \text{plug}(C[I_1 \leq_{\text{int}} I_2]) \)

\( \circ C[! \text{true}] \rightarrow \text{plug}(C[\text{false}]) \)

\( \circ C[! \text{false}] \rightarrow \text{plug}(C[\text{true}]) \)

\( \circ C[\text{true} \&\& B_2] \rightarrow \text{plug}(C[B_2]) \)

\( \circ C[\text{false} \&\& B_2] \rightarrow \text{plug}(C[\text{false}]) \)

\( \circ (C, \sigma)[X = I;] \rightarrow \text{plug}(\langle C, \sigma[I/X]\rangle[[[]]]) \) if \( \sigma(X) \neq \perp \)

\( \circ C[\sigma] S_2 \rightarrow \text{plug}(C[S_2]) \)

\( \circ C[\text{if(true) } S_1 \text{ else } S_2] \rightarrow \text{plug}(C[S_1]) \)

\( \circ C[\text{if(false) } S_1 \text{ else } S_2] \rightarrow \text{plug}(C[S_2]) \)

\( \circ C[\text{while(B) } S] \rightarrow \text{plug}(C[\text{if } (B) \{ S \text{ while(B) } S \} \text{ else }[]]) \)

\( \circ \langle \text{int Xl; } S \rangle \rightarrow \langle S, (Xl \rightarrow 0) \rangle \)

Figure 3.39: \( \mathcal{R}_{\text{RSEC(Imp)}} \) — rewrite logic theory corresponding to the second embedding of the reduction semantics with evaluation contexts of IMP.

\( \circ \text{Cfg} \rightarrow \text{Cfg}' \) if \( \circ \text{plug(split(Cfg))} \rightarrow \text{Cfg}' \)

\( \circ C[\text{Syn}] \rightarrow C[\text{Syn'}] \) if \( C \neq \Box \land \circ \text{Syn} \rightarrow \text{Syn'} \)

\( \circ (C, \sigma)[X] \rightarrow \langle C, \sigma \rangle[[\sigma(X)]] \) if \( \sigma(X) \neq \perp \)

\( \circ I_1 + I_2 \rightarrow I_1 +_{\text{int}} I_2 \)

\( \circ I_1 / I_2 \rightarrow I_1 /_{\text{int}} I_2 \) if \( I_2 \neq 0 \)

\( \circ I_1 <= I_2 \rightarrow I_1 \leq_{\text{int}} I_2 \)

\( \circ \text{!true} \rightarrow \text{false} \)

\( \circ \text{!false} \rightarrow \text{true} \)

\( \circ \text{true} \&\& B_2 \rightarrow B_2 \)

\( \circ \text{false} \&\& B_2 \rightarrow \text{false} \)

\( \circ (C, \sigma)[X = I;] \rightarrow \langle C, \sigma[I/X]\rangle[[[]]] \) if \( \sigma(X) \neq \perp \)

\( \circ \{\} S_2 \rightarrow S_2 \)

\( \circ \text{if(true) } S_1 \text{ else } S_2 \rightarrow S_1 \)

\( \circ \text{if(false) } S_1 \text{ else } S_2 \rightarrow S_2 \)

\( \circ \text{while(B) } S \rightarrow \text{if } (B) \{ S \text{ while(B) } S \} \text{ else }[] \)

\( \circ \langle \text{int Xl; } S \rangle \rightarrow \langle S, (Xl \rightarrow 0) \rangle \)

Figure 3.40: \( \mathcal{R}_{\text{RSEC(Imp)}} \) — rewrite logic theory corresponding to the third embedding of the reduction semantics with evaluation contexts of IMP.
rule of IMP. We also need to add a generic conditional rule, the first one, which completely encapsulates the rewrite logic representation of the splitting/plugging mechanism, so that the language designer can next focus exclusively on the semantic rules rather than on their representation in rewrite logic. The second rewrite rule in discussed, it is optional; if one includes it, as we did, we think that its definition in Figure 3.40 is as simple and natural as it can be. In what regards the remaining rewrite rules, the only perceivable difference between them and their corresponding reduction rules is that they are preceded by ∘.

All the above suggest that, in spite of its apparently advanced context-sensitivity and splitting/plugging mechanism, reduction semantics with evaluation contexts can be safely regarded as a methodological fragment of rewrite logic. Or, put differently, while context-sensitive reduction seems crucial for programming language semantics, it is in fact unnecessary. A conditional rewrite framework can methodologically achieve the same results, and as discussed in this chapter, so can do for the other conventional language semantics approaches.

The following corollary of Theorems 18, 19, and 20 establishes the faithfulness of the representations of the reduction semantics with evaluation contexts of IMP in rewrite logic:

**Corollary 7.** For any IMP configurations C and C′, the following equivalences hold:

\[
\begin{align*}
\text{RSEC(IMP)} \vdash C \rightarrow C' & \iff \text{RSEC(IMP)} \vdash \circ C \rightarrow C' \\
& \iff \text{RSEC(IMP)} \vdash \circ C \rightarrow C' \\
& \iff \text{RSEC(IMP)} \vdash \circ C \rightarrow C'
\end{align*}
\]

and

\[
\begin{align*}
\text{RSEC(IMP)} \vdash C \rightarrow * C' & \iff \text{RSEC(IMP)} \vdash \star C \rightarrow \star C' \\
& \iff \text{RSEC(IMP)} \vdash \star C \rightarrow \star C' \\
& \iff \text{RSEC(IMP)} \vdash \star C \rightarrow \star C'
\end{align*}
\]

Therefore, there is no perceivable computational difference between the reduction semantics with evaluation contexts RSEC(IMP) and its corresponding rewrite logic theories.

**★ Reduction Semantics with Evaluation Contexts of IMP in Maude**

Figure 3.41 shows a Maude representation of the rewrite theory RSEC(IMP) in Figure 3.37 that embeds IMP’s evaluation contexts by making explicit the split/plug mechanism which is implicit in RSEC. Figure 3.41 also includes the Maude definition of configurations (see Figure 3.36).

We took the freedom to implement a simple optimization which works well in Maude, but which may not work as well in other engines or systems (which is why we did not incorporate it as part of the general procedure to represent reduction semantics with evaluation contexts in rewrite logic): we defined the contextual representation operation [\_] to have as result the kind (see Section 2.5) [Syntax] instead of the sort Syntax. This allows us to include the equation plug(Syn) = Syn, where Syn is a variable of sort Syntax, which gives us the possibility to also use terms which do not make use of contexts in the right-hand-sides of rewrite rules. To test the rules for splitting, one can write Maude commands such as the one below, asking Maude to search for all splits of a given term:

\[
\text{search split}(3 \leq (2 + X) / 7) \Rightarrow! \text{Syn}:[\text{Syntax}].
\]

The ! tag on the arrow => in the command above tells Maude to only report the normal forms, in this case the completed splits. As expected, Maude finds all seven splits and outputs the following:

258
mod IMP-CONFIGURATIONS-EVALUATION-CONTEXTS is including IMP-SYNTAX + STATE.
sorts Configuration ExtendedConfiguration.
subsort Configuration < ExtendedConfiguration.
op <_,_> : Stmt State -> Configuration.
op <> : Pgm -> Configuration.
ops (o_) (*_) : Configuration -> ExtendedConfiguration [prec 80]. --- one step
var Cfg Cfg' : Configuration.
crl * Cfg => * Cfg' if o Cfg => Cfg'.
endm

mod IMP-SPLIT-PLUG-EVALUATION-CONTEXTS is including IMP-CONFIGURATIONS-EVALUATION-CONTEXTS.
sorts Syntax Context. subsorts AExp BExp Stmt Configuration < Syntax.
op [] : -> Context. op _[_] : Context Syntax -> [Syntax] [prec 1].
ops split plug : Syntax -> Syntax. --- to split Syntax into context[redex]

var X : Id. var A A1 A2 : AExp. var B B1 B2 : BExp. var S S1 S2 : Stmt.
var Sigma : State. var I1 : Int. var Syn : Syntax. var C : Context.
rl split(Syn) => [][Syn]. eq plug([][Syn]) = Syn. eq plug(Syn) = Syn.

op <_,_> : Context State -> Context. eq plug(< C, Sigma > [Syn]) = < plug(C[Syn]), Sigma >.
crl split(< C, Sigma >) => < C, Sigma > [Syn] if split(S) => C[Syn].

op ___ : Context AExp -> Context. eq plug((C + A2)[Syn]) = plug(C[Syn]) + A2.
crl split(A1 + A2) => (C + A2)[Syn] if split(A1) => C[Syn].

op _-_ : AExp Context -> Context. eq plug((A1 + C)[Syn]) = A1 + plug(C[Syn]).
crl split(A1 + A2) => (A1 + C)[Syn] if split(A2) => C[Syn].

op ___ : Context AExp -> Context. eq plug((C / A2)[Syn]) = plug(C[Syn]) / A2.
crl split(A1 / A2) => (C / A2)[Syn] if split(A1) => C[Syn].

op _-_ : AExp Context -> Context. eq plug((A1 / C)[Syn]) = A1 / plug(C[Syn]).

op ___ : Context AExp -> Context. eq plug((C <= A2)[Syn]) = plug(C[Syn]) <= A2.
crl split(A1 <= A2) => (C <= A2)[Syn] if split(A1) => C[Syn].

op ___ : Int Context -> Context. eq plug((I1 <= C)[Syn]) = I1 <= plug(C[Syn]).
crl split(I1 <= A2) => (I1 <= C)[Syn] if split(A2) => C[Syn].

op !_ : Context -> Context. eq plug((! C)[Syn]) = ! plug(C[Syn]).
crl split(! B) => (! C)[Syn] if split(B) => C[Syn].

crl split(B1 && B2) => (C && B2)[Syn] if split(B1) => C[Syn].

op ___ : Id Context -> Context. eq plug((X = C ;)[Syn]) = X = plug(C[Syn]).
crl split(X = A ;) => (X = C ;)[Syn] if split(A) => C[Syn].

op ___ : Context Stmt -> Context. eq plug((C S2)[Syn]) = plug(C[Syn]) S2.
crl split(S1 S2) => (C S2)[Syn] if split(S1) => C[Syn].

op if(_)_else_ : Context Stmt Stmt -> Context.
crl split(if (B) S1 else S2) => (if (C) S1 else S2)[Syn] if split(B) => C[Syn].
eq plug(if (C) S1 else S2)[Syn] = if (plug(C[Syn])) S1 else S2.
endm

Figure 3.41: The configuration and evaluation contexts of IMP in Maude, as needed for the three variants of reduction semantics with evaluation contexts of IMP in Maude.
Solution 1 (state 1)
states: 8 rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> [[3 <= (2 + X) / 7]

Solution 2 (state 2)
states: 8 rewrites: 19 in ... cpu (. real) (0 rewrites/second)
Syn --> ([3 <= (2 + X) / 7][3]

Solution 3 (state 3)
states: 8 rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= [[])[(2 + X) / 7]

Solution 4 (state 4)
states: 8 rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ([] / 7)[2 + X]

Solution 5 (state 5)
states: 8 rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ([(] + X) / 7))[2]

Solution 6 (state 6)
states: 8 rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= (2 + [)])[X]

Solution 7 (state 7)
states: 8 rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= 2 + X / [])[7]

If, however, we replace any of the rules for splitting with equations, then, as expected, one looses some of the splitting behaviors. For example, if we replace the generic rule for splitting
\[ rl \text{ split}(\text{Syn}) \Rightarrow [[\text{Syn}] \]
by an apparently equivalent equation
\[ eq \text{ split}(\text{Syn}) = [[\text{Syn}] \]
then Maude will be able to detect no other splitting of a term \( t \) except for \( □[t] \) (because Maude executes the equations before the rules; see Section 2.5.6).

Figure 3.42 shows two Maude modules implementing the first two rewrite logic theories \( R_{RSEC(IMP)} \) (Figure 3.38) and \( R_{RSEC(IMP)} \) (Figure 3.39), and Figure 3.43 shows the Maude module implementing the third rewrite theory \( R_{RSEC(IMP)} \) (Figure 3.40), respectively. Each of these three Maude modules imports the module
\[ IMP-CONFIGURATION-EVALUATION-CONTEXTS \] defined in Figure 3.41 and is executable. Maude, through its rewriting capabilities, therefore yields an IMP reduction semantics with evaluation contexts interpreter for each of the three modules in Figures 3.42 and 3.43. For any of them, the Maude rewrite command
\[ \text{rewrite } * < \text{sumPgm} > . \]

where \( \text{sumPgm} \) is the first program defined in the module \( IMP-PROGRAMS \) in Figure 3.4 produces a result of the form (the exact statistics are also irrelevant, so they were replaced by “…”):
\[ \text{rewrites: 42056 in ... cpu (... real) (... rewrites/second)} \]
\[ \text{result ExtendedConfiguration: } * < \text{skip, n } \Rightarrow \theta & \text{s } \Rightarrow 5050 > \]

The reason for this is...

One can use any of the general-purpose tools provided by Maude on the reduction semantics with evaluation contexts definitions above. For example, one can exhaustively search for all possible behaviors of a program using the \texttt{search} command:
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS.
var X : Id . var I I1 I2 : Int . var B B2 : BExp . var S S1 S2 : Stmt .
var Xl : List{Id} . var Sigma : State . var Cfg : Configuration . var C : Context .

\[ \text{crl o } \text{Cfg }\Rightarrow \text{plug}(\langle C,\Sigma \rangle[\Sigma(X)]) \text{ if split(Cfg) }\Rightarrow \langle C,\Sigma \rangle[X] \]
\( \land \) Sigma(X) =/= Bool undefined .
\[ \text{crl o } \text{Cfg }\Rightarrow \text{plug}(C[I1 +Int I2]) \text{ if split(Cfg) }\Rightarrow C[I1 +I2] . \]
\[ \text{crl o } \text{Cfg }\Rightarrow \text{plug}(C[I1 /Int I2]) \text{ if split(Cfg) }\Rightarrow C[I1 /I2] \]
\( \land \) I2 =/= Bool 0 .
\[ \text{crl o } \text{Cfg }\Rightarrow \text{plug}(C[true]) \text{ if split(Cfg) }\Rightarrow C[false] . \]
\[ \text{crl o } \text{Cfg }\Rightarrow \text{plug}(C[if (B) \{S \text{ while }(B) S\} \text{ else }\{\}]) \text{ if split(Cfg) }\Rightarrow C[while (B) S] . \]
\[ \text{rl o } \langle \text{int } Xl ; S \rangle \Rightarrow \langle S, (Xl |\rightarrow 0) \rangle . \]
endm

mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS.
var X : Id . var I I1 I2 : Int . var B B2 : BExp . var S S1 S2 : Stmt .
var Xl : List{Id} . var Sigma : State . var Cfg Cfg' : Configuration . var C : Context .

\[ \text{crl o } \text{Cfg }\Rightarrow \text{Cfg}' \text{ if o split(Cfg) }\Rightarrow \text{Cfg}' . \text{ --- generic rule enabling splitting} \]
\[ \text{crl o } \langle C,\Sigma \rangle[X] \Rightarrow \text{plug}(\langle C,\Sigma \rangle[\Sigma(X)]) \]
if Sigma(X) =/= Bool undefined .
\[ \text{rl o } C[I1 + I2] \Rightarrow \text{plug}(C[I1 +Int I2]) . \]
\[ \text{crl o } C[I1 / I2] \Rightarrow \text{plug}(C[I1 /Int I2]) \]
if I2 =/= Bool 0 .
\[ \text{rl o } C[I1 <= I2] \Rightarrow \text{plug}(C[I1 <=Int I2]) . \]
\[ \text{rl o } C[false] \Rightarrow \text{plug}(C[true]) . \]
\[ \text{rl o } C[true \&\& B2] \Rightarrow \text{plug}(C[B2]) . \]
\[ \text{rl o } C[false \&\& B2] \Rightarrow \text{plug}(C[false]) . \]
\[ \text{rl o } C[\{\} S] \Rightarrow \text{plug}(C[S]) . \]
\[ \text{crl o } \langle C,\Sigma \rangle[X = I ;] \Rightarrow \text{plug}(\langle C,\Sigma[I / X] \rangle[\{\}]) \]
if Sigma(X) =/= Bool undefined .
\[ \text{rl o } C[\{\} S2] \Rightarrow \text{plug}(C[S2]) . \]
\[ \text{rl o } C[if (true) S1 \text{ else } S2] \Rightarrow \text{plug}(C[S1]) . \]
\[ \text{rl o } C[if (false) S1 \text{ else } S2] \Rightarrow \text{plug}(C[S2]) . \]
\[ \text{rl o } C[while (B) S] \Rightarrow \text{plug}(C[if (B) \{S \text{ while }(B) S\} \text{ else }\{\}]) . \]
\[ \text{rl o } \langle \text{int } Xl ; S \rangle \Rightarrow \langle S, (Xl |\rightarrow 0) \rangle . \]
endm

Figure 3.42: The first two reduction semantics with evaluation contexts of IMP in Maude.
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS.

var X : Id . var I I1 I2 : Int . var B B2 : BExp . var S S1 S2 : Stmt . var Xl : List{Id} . var Sigma : State . var Cfg Cfg' : Configuration . var Syn Syn' : Syntax . var C : Context .

crl o Cfg => Cfg' if plug(o split(Cfg)) => Cfg' . --- generic rule enabling splitting
crl o C[Syn] => C[Syn'] if C =/=Bool [] /\ o Syn => Syn' . --- characteristic rule

crl o < C,Sigma >[X] => < C,Sigma >[Sigma(X)]
  if Sigma(X) =/=Bool undefined .
rl o I1 + I2 => I1 +Int I2 .
crl o I1 / I2 => I1 /Int I2
  if I2 =/=Bool 0 .
rl o I1 <= I2 => I1 <=Int I2 .
rl o ! true => false .
rl o ! false => true .
rl o false && B2 => false .
rl o {S} => S .
crl o < C,Sigma >[X = I ;] => < C,Sigma[I / X] >[{}]
  if Sigma(X) =/=Bool undefined .
rl o {} S2 => S2 .
rl o if (true) S1 else S2 => S1 .
rl o if (false) S1 else S2 => S2 .
rl o while (B) S => if (B) {S while (B) S} else {} .
rl o < int X1 ; S > => < S, (X1 |-> 0) > .
endm

Figure 3.43: The third reduction semantics with evaluation contexts of IMP in Maude.
As expected, only one behavior will be discovered because our IMP language so far is deterministic. Not unexpectedly, the same number of states as in the case of small-step SOS and MSOS will be discovered by this search command, namely 1709. Indeed, the splitting/cooling mechanism of RSEC is just another way to find where the next reduction step should take place; it does not generate any different reductions of the original configuration.

3.7.4 Notes

Reduction semantics with evaluation contexts was introduced by Felleisen and his collaborators (see, e.g., [25], [88]) as a variant small-step structural operational semantics. By making the evaluation context explicit and modifiable, reduction semantics with evaluation contexts is considered by many to be a significant improvement over small-step SOS. Like small-step SOS, reduction semantics with evaluation contexts has been broadly used to give semantics to programming languages and to various calculi. We here only briefly mention some strictly related work.

How expensive is the splitting of a term into an evaluation context and a redex? Unfortunately, it cannot be more efficient than testing the membership of a word to a context-free grammar and the latter is expected to be cubic in the size of the original term (folklore). Indeed, consider $G$ an arbitrary CFG whose start symbol is $S$ and let $G_C$ be the “evaluation context” CFG grammar adding a fresh “context” nonterminal $C$, a fresh terminal #, and productions $C \rightarrow □ | CS#$. Then it is easy to see that a word $α$ is in the language of $G$ if and only if $#α#$ can be split as a contextual representation (can only be $(□α#)(#)$). Thus, we should expect, in the worst case, a cubic complexity to split a term into an evaluation context and a redex. An additional exponent needs to be added, thus making splitting expected to be a quadratic operation in the worst case, when nested contexts are allowed in rules (i.e., when the redex is itself a contextual representation). Unfortunately, this terrible complexity needs to be paid at each step of reduction, not to mention that the size of the program to reduce can also grow as it is reduced. One possibility to decrease this complexity is to attempt to incrementally compute at each step the evaluation context that is needed at the next step (like in refocusing; see below); however, in the worst case the right-hand-sides of rules may contain no contexts, in which case a fresh split is necessary at each step.

Besides our own efforts, we are aware of three other attempts to develop executable engines for reduction semantics with evaluation contexts, which we discuss here in chronological order:

1. A specification language for syntactic theories with evaluation contexts is proposed by Xiao et al. [90], together with a system which generates Ocaml interpreters from specifications. Although the compiler in [90] is carefully engineered, as rightfully noticed by Danvy and Nielsen in [20] it cannot avoid the quadratic overhead due to the context-decomposition step. This is consistent with our own observations expressed at several places in this section, namely that the advanced parsing underlying reduction semantics with evaluation contexts is the most expensive part when one is concerned with execution. Fortunately, the splitting of syntax into context and redex can be and typically is taken for granted in theoretical developments, making abstraction of the complexity of its implementation.

2. A technique called refocusing is proposed by Danvy and Nielsen in [20], [19]. The idea underlying refocusing is to keep the program decomposed at all times (in a first-order continuation-like form) and to perform minimal changes to the resulting structure to find the next redex. Unfortunately, refocusing appears to work well only with restricted RSEC definitions, namely ones whose evaluation contexts grammar has the property of unique decomposition of a term into a context and a redex (so constructs like the non-deterministic addition of IMP are disallowed), and whose reduction rules are deterministic.
3. PLT-Redex, which is implemented in Scheme by Findler and his collaborators \cite{38, 24}, is perhaps the most advanced tool developed specifically to execute reduction semantics with evaluation contexts. PLT-Redex builds upon a direct implementation of context-sensitive reduction, so it cannot avoid the worst-case quadratic complexity of context decomposition, same as the interpreters generated by the system in \cite{90, 89} discussed above. Several large language semantics engineering case studies using PLT-Redex are discussed in \cite{24}.

Our embeddings of reduction semantics with evaluation contexts into rewrite logic are inspired from a related embedding by Şerbănütă et al. in \cite{74}. The embedding in \cite{74} was similar to our third embedding here, but it included splitting rules also for terms in reducible form, e.g., \(\text{split}(I_1 \leq I_2) \rightarrow \Box[I_1 \leq I_2]\). Instead, we preferred to include a generic rule \(\text{split}(\text{Syn}) \rightarrow \Box[\text{Syn}]\) here, which allows us to more mechanically derive the rewrite rules for splitting from the CFG of evaluation contexts. Calculating the exact complexity of our approach seems to be hard, mainly because of optimizations employed by rewrite engines, e.g., indexing. Since at each step we still search for all the relevant splits of the term into an evaluation context and a redex, in the worst case we still pay the quadratic complexity. However, as suggested by the performance numbers in \cite{74} comparing Maude running the resulting rewrite theory against PLT-Redex, which favor the former by a large margin, our embeddings may serve as alternative means to getting more efficient implementations of reduction semantics engines. There are strong reasons to believe that our third embedding can easily be automated in a way that the user never sees the split/plug operations.

### 3.7.5 Exercises

**Exercise 127.** Suppose that one does not like mixing semantic components with syntactic evaluation contexts as we did above (by including the production \(\text{Context} ::= \langle \text{Context}, \text{State} \rangle\)). Instead, suppose that one prefers to work with configuration tuples like in SOS, holding the various components needed for the language semantics, the program or fragment of program being just one of them. In other words, suppose that one wants to make use of the contextual representation notation only on the syntactic component of configurations. In this case, the characteristic rule becomes

\[
\langle e, \gamma \rangle \rightarrow \langle e', \gamma' \rangle
\]

\[
\langle c[e], \gamma \rangle \rightarrow \langle c[e'], \gamma' \rangle
\]

where \(\gamma\) and \(\gamma'\) consist of configuration semantic components that are necessary to evaluate \(e\) and \(e'\), respectively, such as states, outputs, stacks, etc. Modify accordingly the six reduction semantics with evaluation contexts rules discussed at the beginning of Section 3.7.

The advantage of this approach is that it allows the evaluation contexts to be defined exclusively over the syntax of the language. However, configurations holding code and state still need to be defined. Moreover, many rules which looked compact before, such as \(i_1 \leq i_2 \rightarrow i_1 \leq i_0, i_2\), will now look heavier, e.g., \(\langle i_1 \leq i_2, \sigma \rangle \rightarrow \langle i_1 \leq i_2, \sigma \rangle\).

**Exercise 128.** Like in Exercise 127, suppose that one does not like to mix syntactic and semantic components in evaluation contexts, but that, instead, one is willing to accept to slightly enrich the syntax of the programming language with a special statement construct “\(\text{Stmt} ::= \text{int } Id = \text{AExp;}\)”, which both declares and initializes a variable\(^\text{10}\). Then

1. Write structural identities that desugar the current top-level program variable declarations \(\text{int } x_1, ..., x_n; \ s\) into statements of the form \(\text{int } x_1 = 0; \ldots \text{int } x_n = 0; \ s\).

\(^{10}\) Similar language constructs exist in many programming language (C, Java, etc.).
2. Add a new context production that allows evaluation after the new variable declarations.

3. Modify the variable lookup and assignment rules discussed above so that one uses the new declarations instead of a state. Hint: the context should have the form “\texttt{int }x=i; \ c\texttt{”}.

The advantage of this approach is that one does not need an explicit state anymore, so the resulting definition is purely syntactic. In fact, the state is there anyway, but encoded syntactically as a sequence of variable initializations preceding any other statement. This trick works in this case, but it cannot be used as a general principle to eliminate configurations in complex languages.

Exercise* 129. Exercise 127 suggests that one can combine MSOS (Section 3.6) and evaluation contexts, in that one can use MSOS’s labels to obtain modularity at the configuration level and one can use the evaluation contexts idea to detect and modify the contexts/redexes in the syntactic component of a configuration. Rewrite the six rules discussed at the beginning of Section 3.7 as they would appear in a hypothetical framework merging MSOS and evaluation contexts.

Exercise 130. Modify the reduction semantics with evaluation contexts of IMP in Figures 3.30 and 3.31 so that \texttt{/} short-circuits when its numerator evaluates to 0.

Hint: Make \texttt{/} strict in only the first argument, then use a rule to reduce 0 \texttt{/} \texttt{a}2 to 0 and a rule to reduce \texttt{i}1 \texttt{/} \texttt{a}2 to \texttt{i}1 /’ \texttt{a}2 when \texttt{i}1 \neq 0, where /’ is strict in its second argument, and finally a rule to reduce \texttt{i}1 /’ \texttt{i}2 to \texttt{i}1 /’\texttt{a}2 when \texttt{i}2 \neq 0.

Exercise 131. Modify the reduction semantics with evaluation contexts of IMP in Figures 3.30 and 3.31 so that conjunction is not short-circuit anymore but, instead, is non-deterministically strict in both its arguments.

Exercise 132. Give an alternative reduction semantics of IMP with evaluation contexts following the approach in Exercise 127 (that is, use evaluation contexts only for the IMP language syntax, and handle the semantic components using configurations, like in SOS).

Exercise 133. Give an alternative reduction semantics of IMP with evaluation contexts following the approach in Exercise 128.

Exercise* 134. Give a semantics of IMP using the hypothetical framework combining reduction semantics with evaluation contexts and MSOS proposed in Exercise 129.

Exercise 135. Modify the rewrite theory \(\mathcal{R}_{\text{RSEC IMP}}\) in Figure 3.37 so that later on one can define the reduction semantics of \texttt{/} to short-circuit when the numerator evaluates to 0 (as required in Exercises 137, 143 and 149).

Exercise 136. Modify the rewrite theory \(\mathcal{R}_{\text{RSEC IMP}}\) in Figure 3.37 so that one can later on define the reduction semantics of conjunction to be non-deterministically strict in both its arguments (as required in Exercises 138, 144 and 150).

Exercise 137. Modify the rewrite theory \(\mathcal{R}_{\text{RSEC IMP}}\) in Figure 3.38 to account for the reduction semantics of \texttt{/} that short-circuits when the numerator evaluates to 0 (see also Exercise 135).

Exercise 138. Modify the rewrite theory \(\mathcal{R}_{\text{RSEC IMP}}\) in Figure 3.38 to account for the reduction semantics of conjunction that defines it as non-deterministically strict in both its arguments (see also Exercise 136).
Exercise 139. As discussed in several places so far in Section 3.7, the reduction semantics rules for variable lookup and assignment can also be given in a way in which their left-hand-side terms are not in contextual representation (i.e., ⟨c[x], σ⟩ instead of ⟨c, σ⟩[x], etc.). Modify the corresponding rewrite rules of $R_{RSEC(IMP)}$ in Figure 3.38 to account for this alternative reduction semantics.

Exercise 140. Modify the rewrite logic theory $R_{RSEC(IMP)}$ in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of $IMP$ in Exercise 132.

Exercise 141. Modify the rewrite logic theory $R_{RSEC(IMP)}$ in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of $IMP$ in Exercise 133.

Exercise* 142. Combining the underlying ideas of the embedding of MSOS in rewrite logic discussed in Section 3.6.3 and the embedding of reduction semantics with evaluation contexts in Figure 3.33, give a rewrite logic semantics of $IMP$ corresponding to the semantics of $IMP$ in Exercise 134.

Exercise 143. Same as Exercise 137 but for $R_{RSEC(IMP)}$ in Figure 3.39 (instead of $R_{RSEC(IMP)}$).

Exercise 144. Same as Exercise 138 but for $R_{RSEC(IMP)}$ in Figure 3.39 (instead of $R_{RSEC(IMP)}$).

Exercise 145. Same as Exercise 139 but for $R_{RSEC(IMP)}$ in Figure 3.39 (instead of $R_{RSEC(IMP)}$).

Exercise 146. Same as Exercise 140 but for $R_{RSEC(IMP)}$ in Figure 3.39 (instead of $R_{RSEC(IMP)}$).

Exercise 147. Same as Exercise 141 but for $R_{RSEC(IMP)}$ in Figure 3.39 (instead of $R_{RSEC(IMP)}$).

Exercise* 148. Same as Exercise 142 but for Figure 3.39 (instead of Figure 3.38).

Exercise 149. Same as Exercise 137 but for $R_{RSEC(IMP)}$ in Figure 3.40 (instead of $R_{RSEC(IMP)}$).

Exercise 150. Same as Exercise 138 but for $R_{RSEC(IMP)}$ in Figure 3.40 (instead of $R_{RSEC(IMP)}$).

Exercise 151. Same as Exercise 139 but for $R_{RSEC(IMP)}$ in Figure 3.40 (instead of $R_{RSEC(IMP)}$).

Exercise 152. Same as Exercise 140 but for $R_{RSEC(IMP)}$ in Figure 3.40 (instead of $R_{RSEC(IMP)}$).

Exercise 153. Same as Exercise 141 but for $R_{RSEC(IMP)}$ in Figure 3.40 (instead of $R_{RSEC(IMP)}$).

Exercise* 154. Same as Exercise 142 but for Figure 3.40 (instead of Figure 3.38).

Exercise 155. Modify the Maude code in Figures 3.41 and 3.42 so that / short-circuits when its numerator evaluates to 0 (see also Exercises 130, 135, 137, 143, and 149).

Exercise 156. Modify the Maude code in Figures 3.41 and 3.42 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments (see also Exercises 131, 136, 138, 144, and 150).

Exercise 157. Modify the Maude code in Figures 3.41 and 3.42 to account for the alternative reduction semantics in Exercises 139, 145, and 151.
Exercise 158. Modify the Maude code in Figures 3.41 and 3.42 to account for the alternative reduction semantics in Exercises 140, 146, and 152.

Exercise 159. Modify the Maude code in Figures 3.41 and 3.42 to account for the alternative reduction semantics in Exercises 141, 147, and 153.

Exercise* 160. Modify the Maude code in Figures 3.41 and 3.42 to account for the semantics in Exercises 142, 148, and 154.

Exercise 161. Same as Exercise 86 but for reduction semantics with evaluation contexts instead of small-step SOS: add variable increment to IMP, like in Section 3.7.2.

Exercise 162. Same as Exercise 90 but for reduction semantics with evaluation contexts instead of small-step SOS: add input/output to IMP, like in Section 3.7.2.

Exercise* 163. Consider the hypothetical framework combining MSOS with reduction semantics with evaluation contexts proposed in Exercise 129 and in particular the IMP semantics in such a framework in Exercise 134 its rewrite logic embeddings in Exercises 142, 148, and 154 and their Maude implementation in Exercise 160. Define the semantics of the input/output constructs above modularly first in the framework in discussion, then using the rewrite logic embeddings, and finally in Maude.

Exercise 164. Same as Exercise 95 but for reduction semantics with evaluation contexts instead of small-step SOS: add abrupt termination to IMP, like in Section 3.7.2.

Exercise 165. Same as Exercise 104 but for reduction semantics with evaluation contexts instead of small-step SOS: add dynamic threads to IMP, like in Section 3.7.2.

Exercise 166. Same as Exercise 109 but for reduction semantics with evaluation contexts instead of small-step SOS: add local variables using let to IMP, like in Section 3.7.2.

Exercise* 167. This exercise asks to define IMP++ in reduction semantics, in various ways. Specifically, redo Exercises 114, 115, 116, 117, and 118 but for the reduction semantics with evaluation contexts of IMP++ discussed in Section 3.7.2 instead of its small-step SOS in Section 3.5.6.
Try to get rid of kinds in the Maude implementation, so that I don’t need to introduce them in the EL, RL and Maude sections.

should I include the paragraph below?

Like for the first embedding, one can now use a context-insensitive rewrite engine, in particular its search capabilities, to explore computations in the original context reduction definition. For example, if one is interested in listing all terms that are reachable from a given term t using the original context reduction definition, then all one needs to do is to invoke a search procedure on the term \{split(t)\} and limit its responses to those matching the pattern \{□[Syntax]\}. Moreover, if one is only interested in the values that a program t can reduce to, then, assuming that one defined a syntactic category Value, all one needs to do is to filter the results to those matching the pattern \{□[Value]\}. Even more interestingly, if one is interested in all those reachable configurations in which x = 0 and y = z, then, assuming that State is defined as a comma-separated set of pairs variable \mapsto value, all one needs to do is to limit the results of the search to those terms matching the pattern \{□[\langle Syntax, (x \mapsto 0, y \mapsto v, z \mapsto v, State)\rangle]\}. However, unlike for the first embedding, with this second embedding one cannot blindly let the rewrite engine execute context reduction definitions. That is because the rewrite engine may choose to stop the rewriting of the term \{split(t)\} after each application of a rule of the form rule \(i\), by choosing to apply the rule \(\text{split}(N) \rightarrow □[N]\) (see Figure ??).

should we have a more indepth discussion on the complexity of the various techniques?

Narciso: it would be good to add numbers comparing plt-redex with Maude on context reduction definitions.

G: it would be good to add numbers everywhere, in all semantic styles, comparing the various realizations in various languages.

make sure the exercise scheme is consistent everywhere; for example, there is no exercise on generating error at div by 0 in RSEC, and there are no exercises in the RL and Maude subsections of the previous semantics

make the point that rsec separates concerns when compared to sos: it separates defining evaluation strategies from the actual reduction steps
3.8 The Chemical Abstract Machine (CHAM)

The chemical abstract machine, or the CHAM, is both a model of concurrency and a specific operational semantics style. The states of a CHAM are metaphorically regarded as chemical solutions formed with floating molecules. Molecules can interact with each other by means of reactions. A reaction can involve several molecules and can change them, delete them, and/or create new molecules. One of the most appealing aspects of the chemical abstract machine is that its reactions can take place concurrently, unrestricted by context. To facilitate local computation and to represent complex data-structures, molecules can be nested by encapsulating groups of molecules as sub-solutions. The chemical abstract machine was proposed as an alternative to SOS and its variants, including reduction semantics with evaluation contexts, in an attempt to circumvent their limitations, particularly their lack of support for true concurrency.

CHAM Syntax

The basic molecules of a CHAM are ordinary algebraic terms over a user-defined syntax. Several molecules wrapped within a membrane form a solution, which is also a molecule. The CHAM uses the symbols ¦ and ¦ to denote membranes. For example, \[\{m_1 \ m_2 \ldots \ m_k\}\] is a solution formed with the molecules \(m_1, m_2, \ldots, m_k\). The order of molecules in a solution is irrelevant, so a solution can be regarded as a multi-set (or bag) of molecules wrapped within a membrane. Since solutions are themselves molecules, we can have arbitrarily nested molecules. This nesting mechanism is generic for all CHAMs and has the following (algebraic) syntax:

\[
\begin{align*}
\text{Molecule} & \ ::= \ \text{Solution} \mid \text{Molecule} \triangleleft \text{Solution} \\
\text{Solution} & \ ::= \ \{\text{Bag}\{\text{Molecule}\}\}
\end{align*}
\]

The operator \(\triangleleft\) is called the airlock operator and will be discussed shortly (under general CHAM laws), after we discuss the CHAM rules. When defining a CHAM, one is only allowed to extend the syntax of molecules, which implicitly also extends the syntax that the solution terms can use. However, one is not allowed to explicitly extend the syntax of solutions. In other words, solutions can only be built using the generic syntax above, on top of user-defined syntactic extensions of molecules. Even though we do not formalize it here (and we are not aware of other formulations elsewhere either), it is understood that one can have multiple types of molecules in a CHAM.

Specific CHAM Rules

In addition to extending the syntax of molecules, a CHAM typically also defines a set of rules, each rule being a rule schemata but called a rule for simplicity. A CHAM rule has the form

\[m_1 \ m_2 \ldots \ m_k \rightarrow m'_1 \ m'_2 \ldots \ m'_l\]

where \(m_1, m_2, \ldots, m_k\) and \(m'_1, m'_2, \ldots, m'_l\) are not necessarily distinct molecules (since CHAM rules are schemata, these molecule terms may contain meta-variables). Molecules appearing in a rule are restricted to contain only subsolution terms which are either solution meta-variables or otherwise have the form \([m]\), where \(m\) is some molecule term. For example, a CHAM rule cannot contain subsolution terms of the form \([m \ s]\), \([m_1 \ m_2]\), or \([m_1 \ m_2 \ s]\), with \(m, m_1, m_2\) molecule terms and \(s\) solution term, but it can contain ones of the form \([m]\), \([m_1 \triangleleft [m_2]\]\), \([m \triangleleft s]\), etc. This restriction is justified by chemical intuitions, namely that matching inside a solution is a rather complex operation which needs special handling (the airlock operator \(\triangleleft\) is used for this purpose). Note that CHAM rules are unconditional, that is, they have no premises.
General CHAM Laws

Any chemical abstract machine obeys the four laws below. Let CHAM be a chemical abstract machine. Below we assume that $mol$, $mol'$, $mol_1$, etc., are arbitrary concrete molecules of CHAM (i.e., no meta-variables) and that $sol$, $sol'$, etc., are concrete solutions of it. If $sol$ is the solution $\{mol_1 \ mol_2 \ldots \ mol_k\}$ and $sol'$ is the solution $\{mol'_1 \ mol'_2 \ldots \ mol'_l\}$, then $sol \uplus sol'$ is the solution $\{mol_1 \ mol_2 \ldots \ mol_k \ mol'_1 \ mol'_2 \ldots \ mol'_l\}$.

1. **The Reaction Law.** Given a CHAM rule

   $m_1 \ m_2 \ldots \ m_k \rightarrow m'_1 \ m'_2 \ldots \ m'_l$ ∈ CHAM

   if $mol_1, mol_2, \ldots, mol_k$ and $mol'_1, mol'_2, \ldots, mol'_l$ are (concrete) instances of $m_1 \ m_2 \ldots \ m_k$ and of $m'_1 \ m'_2 \ldots \ m'_l$ by a common substitution, respectively, then

   $CHAM \vdash \{mol_1 \ mol_2 \ldots \ mol_k\} \rightarrow \{mol'_1 \ mol'_2 \ldots \ mol'_l\}$

2. **The Chemical Law.** Reactions can be performed freely within any solution:

   $CHAM \vdash sol \rightarrow sol'$

   $CHAM \vdash \{mol \} \uplus \{sol\} \rightarrow \{mol \} \uplus \{sol'\}$

3. **The Membrane Law.** A subsolution can evolve freely in any solution context $\{\text{ctx}[\square]\]}$:

   $CHAM \vdash sol \rightarrow sol'$

   $\vdash \{\text{ctx}[sol]\} \rightarrow \{\text{ctx}[sol']\}$

4. **The Airlock Law.**

   $CHAM \vdash \{mol\} \uplus \{sol\} \leftrightarrow \{mol \ uplus sol\}$

   Note the unusual fact that $m_1 \ m_2 \ldots \ m_k \rightarrow m'_1 \ m'_2 \ldots \ m'_l$ being a rule in CHAM does not imply that $CHAM \vdash m_1 \ m_2 \ldots \ m_k \rightarrow m'_1 \ m'_2 \ldots \ m'_l$. Indeed, the CHAM rules are regarded as descriptors of changes that can take place in solutions and only in solutions, while CHAM sequents are incarnations of those otherwise purely abstract rules. What may be confusing is that the same applies also when $k$ and $l$ (the numbers of molecules in the left-hand and right-hand-sides of the CHAM rule) happen to be 1 and $m_1$ and $m'_1$ happen to be solutions that contain no meta-variables. The two $\rightarrow$ arrows, namely the one in CHAM rules and the one in CHAM sequents, ought to be different symbols; however, we adhere to the conventional CHAM notation which uses the same symbol for both. Moreover, when the CHAM is clear from context, we also follow the conventional notation and drop it from sequents, that is, we write $sol \rightarrow sol'$ instead of $CHAM \vdash sol \rightarrow sol'$. While we admit that these conventions may sometimes be confusing, in that $sol \rightarrow sol'$ can be a rule or a sequent or even both, we hope that the context makes it clear which one is meant.

The Reaction Law says that CHAM rules can only apply in solutions (wrapped by a membrane), and not arbitrarily wherever they match. The Chemical Law says that once a reaction take place in a certain solution, it can take place in any other larger solution. In other words, the fact that a solution has more molecules than required by the rule does not prohibit the rule from applying. The Reaction and the Chemical laws together say that CHAM rules can apply inside any solutions having some molecules that match the left-hand side

---

11To avoid inventing new names, it is common to use CHAM both as an abbreviation for “the chemical abstract machine” and as a name of an arbitrary but fixed chemical abstract machine.
of the CHAM rule. An interesting case is when the left-hand-side term of the CHAM rule has only one molecule, i.e., when \( k = 1 \), because the CHAM rule is still allowed to only apply within a solution; it cannot apply in other places where the left-hand-side happens to match.

The Membrane Law says that reactions can take place in any solution context. Indeed, \([\text{ctxt}[\text{sol}]]\) says that the solution \( \text{sol} \) (which is wrapped in a membrane) appears somewhere, anywhere, inside a solution context \([\text{ctxt}[\square]]\). Here \( \text{ctxt} \) can be any bag-of-molecule context, and we write \( \text{ctxt}[\square] \) to highlight the fact that it is a context with a hole \( \square \). By wrapping \( \text{ctxt}[\square] \) in a membrane we enforce a solution context. This rule also suggests that, at any given moment, the global term to rewrite using the CHAM rules should be a solution. Indeed, the CHAM rewriting process gets stuck as soon as the term becomes a proper molecule (not a solution), because the Membrane Law cannot apply.

The Airlock Law is reversible (i.e., it comprises two rewrite rules, one from left-to-right and one from right-to-left) and it allows to extract a molecule from a solution, putting the rest of the solution within a membrane. Using this law one can, for example, rewrite a solution \([\text{mol}_1 \text{mol}_2 \ldots \text{mol}_k]\) into \([\text{mol}_1 \leftrightarrow \{\text{mol}_2 \ldots \text{mol}_k\}]\). The advantage of doing so is that one can now match the molecule \( \text{mol}_1 \) within other rules. Indeed, recall that sub-solutions that appear in rules cannot specify any particular molecule term among the rest of the solution, unless the solution contains precisely that molecule. Since \( \text{mol}_1 \leftrightarrow \{\text{mol}_2 \ldots \text{mol}_k\} \) is a molecule, \( \{\text{mol}_2 \ldots \text{mol}_k\} \) can match molecule terms of the form \( \{m \leftrightarrow s\} \) appearing in CHAM rules, this way one effectively matching (and possibly modifying) the molecule \( \text{mol}_1 \) via the specific CHAM rules. The Airlock Law is the only means provided by the CHAM to extract or put molecules in a solution.

The four laws above do not completely define the CHAM rewriting; they are only properties that the CHAM rewriting should satisfy. In particular, they do not capture the concurrency potential of the CHAM.

**Definition 23.** The four laws above give us a proof system for CHAM sequents. As usual, CHAM \( \vdash \text{sol} \rightarrow \text{sol}' \) in isolation means that it is derivable. Also, let \( \rightarrow^* \) denote the reflexive and transitive closure of \( \rightarrow \), that is, CHAM \( \vdash \text{sol} \rightarrow^* \text{sol}' \) iff \( \text{sol} = \text{sol}' \) or there is some \( \text{sol}'' \) such that CHAM \( \vdash \text{sol} \rightarrow \text{sol}'' \) and CHAM \( \vdash \text{sol}'' \rightarrow^* \text{sol}' \). Finally, CHAM \( \vdash \text{sol} \leftrightarrow \text{sol}' \) is a shorthand for the sequents CHAM \( \vdash \text{sol} \rightarrow \text{sol}' \) and CHAM \( \vdash \text{sol}' \rightarrow \text{sol} \), and we say that it is derivable iff the two sequents are derivable.

None of the two sequents in Definition 23 captures the underlying concurrent computation of the CHAM. Indeed, CHAM \( \vdash \text{sol} \rightarrow \text{sol}' \) says that one and only one reaction takes place somewhere in \( \text{sol} \), while CHAM \( \vdash \text{sol} \rightarrow^* \text{sol}' \) says that arbitrarily many steps take place, including ones which can be done concurrently but also ones which can only take place sequentially. Therefore, we can think of the four laws above, and implicitly of the sequents CHAM \( \vdash \text{sol} \rightarrow \text{sol}' \) and CHAM \( \vdash \text{sol} \rightarrow^* \text{sol}' \), as expressing the descriptive capability of the CHAM: what is possible and what is not possible to compute using the CHAM, and not how it operates. Nevertheless, it is recommended to think of CHAM reactions as taking place concurrently whenever they do not involve the same molecules, even though this “concurrent reaction” notion is not formalized here. We are actually not aware of any works that formalize the CHAM concurrency.

A common source of misunderstanding the CHAM is to wrongly think of CHAM rules as ordinary rewriting rules modulo the associativity, commutativity and identity of the molecule grouping (inside a solution) operation. The major distinction between CHAM rules and such rewrite rules is that the former only apply within solutions (i.e., wrapped by membranes) no matter whether the rule contains one or more molecules in its left-hand or right-hand terms, while the latter apply anywhere they match. For example, supposing that we extend the syntax of molecules with the syntax of IMP in Section 3.1.1 and add a CHAM rule \( m + 0 \rightarrow m \), then we can rewrite the solution \([3 + 0] 7\) to solution \([3 \ 7]\), but we cannot rewrite the molecule \( 5 / (3 + 0) \) to molecule \( 5 / 3 \) regardless of what context it is in, because \( 3 / 0 \) is not in a solution. We cannot even rewrite the isolated (i.e., not in a solution context) term \( 3 + 0 \) to \( 3 \) in CHAM, for the same reason.
Classification of CHAM Rules

The rules of a CHAM are typically partitioned into three intuitive categories, namely heating, cooling and reaction rules, although there are no formal requirements imposing a rule to be into one category or another. Moreover, the same laws discussed above apply the same way to all categories of rules, and the same restrictions preventing multiset matching apply to all of them.

- **Heating** rules, distinguished by using the relation symbol $\leftarrow$ instead of $\rightarrow$, are used to structurally rearrange the solution so that reactions can take place.

- **Cooling** rules, distinguished by using the relation symbol $\leftarrow$ instead of $\rightarrow$, are used after reactions take place to structurally rearrange the solution back into a convenient form, including to remove useless molecules or parts of them.

- **Reaction** rules, which capture the intended computational steps and use the conventional rewrite symbol $\rightarrow$, are used to evolve the solution in an irreversible way.

The heating and cooling rules can typically be paired, with each heating rule $l \leftarrow r$ having a symmetric cooling rule $l \leftarrow r$, so that we can view them as a single bidirectional heating/cooling rule. The CHAM notation for writing such heating/cooling rules is the following:

$$l \Leftrightarrow r$$

In particular, it makes sense to regard the airlock axiom as an example of such a heating/cooling bidirectional rule, that is,

$$\{m_1 \ m_2 \ldots \ m_k\} \Leftrightarrow \{m_1 \ \{m_2 \ldots \ m_k\}\}$$

where $m_1$, $m_2$, $\ldots$, $m_k$ are molecule meta-variables. The intuition here is that we can heat the solution to extract $m_1$ in an airlock, or we can cool it down so that the airlock $m_1$ is diffused within the solution. However, we need to assume one such rule for each $k > 0$.

As one may expect, the reaction rules are the heart of the CHAM and properly correspond to state transitions. The heating and cooling rules express structural rearrangements, so that the reaction rules can match and apply. In other words, we can view the reaction rules as being applied modulo the heating and cooling rules. We are going to suggestively use the notation $\text{CHAM} \vdash sol \rightarrow sol'$, respectively $\text{CHAM} \vdash sol \leftarrow sol'$ whenever the rewrite step taking $sol$ to $sol'$ is a heating rule, respectively a cooling rule. Similarly, we may use the notations $\text{CHAM} \vdash sol \rightarrow^* sol'$ and $\text{CHAM} \vdash sol \leftarrow^* sol'$ for the corresponding reflexive/transitive closures. Also, to emphasize the fact that there is only one reaction rule applied, we take the freedom to (admittedly ambiguously) write $\text{CHAM} \vdash sol \rightarrow sol'$ instead of $\text{CHAM} \vdash sol(\rightarrow \cup \leftarrow \cup \rightarrow)^*sol'$ whenever all the involved rules but one are heating or cooling rules.

3.8.1 The CHAM of IMP

We next show how to give IMP a CHAM semantics. CHAM is particularly well-suited to giving semantics to concurrent distributed calculi and languages, yielding considerably simpler definitions than those afforded by SOS. Since IMP is sequential, it cannot take full advantage of the CHAM’s true concurrency capabilities; the multi-threaded IMP++ language discussed in Section 3.5 will make better use of CHAM’s capabilities. Nevertheless, some of CHAM’s capabilities turn out to be useful even in this sequential language application, others turn out to be deceiving. Our CHAM semantics for IMP below follows in principle the reduction semantics with evaluation contexts definition discussed in Section 3.7.1. One can formally show that a step
performed using reduction under evaluation contexts is equivalent to a suite of heating steps, followed by one reaction step, followed by a suite of cooling steps.

The CHAM defined below is just one possible way to give IMP a CHAM semantics. CHAM, like rewriting, is a general framework which does not impose upon its users any particular definitional style. In our case, we chose to conceptually distinguish two types of molecules; we say “conceptually” because, for simplicity, we prefer to define only one Molecule syntactic category in our CHAM:

- **Syntactic molecules**, which include all the syntax of IMP in Section 3.1.1 plus all the syntax of its evaluation contexts in Section 3.7.1 plus a mechanism to flatten evaluation contexts; for simplicity, we prefer to not include a distinct type of molecule for each distinct syntactic category of IMP.

- **State molecules**, which are pairs $x \mapsto i$, where $x \in Id$ and $i \in Int$.

For clarity, we prefer to keep the syntactic and the state molecules in separate solutions. More precisely, we work with top-level configurations which are solutions of the form

$$\{|\{Syntax\}| \{State\}\}$$

Syntax and State are solutions containing syntactic and state molecules, respectively. For example,

$$\{|\{x = 3 / (x + 2)\}; \{x \mapsto 1 \ y \mapsto 0\}\}$$

is a CHAM configuration containing the statement “$x = 3 / (x + 2)$” and state “$x \mapsto 1, y \mapsto 0$”.

The state molecules and implicitly the state solution are straightforward. State molecules are not nested and state solutions are simply multisets of molecules of the form $x \mapsto i$. The CHAM does not allow us to impose constraints on solutions, such as that the molecules inside the state solution indeed define a partial function and not some arbitrary relation (i.e., there is at most one molecule $x \mapsto i$ for each $x \in Id$). Instead, the state solution will be used in such a way that the original state solution will embed a proper partial function and each rule will preserve this property. For example, the CHAM rule for variable assignment, say when assigning integer $i$ to variable $x$, will rewrite the state molecule from $\{|x \mapsto j \cdot \emptyset|\}$ to $\{|x \mapsto i \cdot \emptyset|\}$.

The top level syntactic solution holds the current program or fragment of program that is still left to be processed. It is not immediately clear how the syntactic solution should be represented in order to be able to give IMP a CHAM semantics. The challenge here is that the IMP language constructs have evaluation strategies and the subterms that need to be next processed can be arbitrarily deep into the program or fragment of program, such as the framed $x$ in “$x = 3 / (x + 2)$”;”.

Failed attempts to represent syntax. A natural approach to represent the syntax of a programming language in CHAM may be to try to use CHAM’s heating/cooling and molecule/solution nesting mechanisms to decompose syntax unambiguously in such a way that the redex (i.e., the subterm which can be potentially reduced next; see Section 3.7) appears as a molecule in the top syntactic solution. That is, if $p = c[t]$ is a program or fragment of program which can be decomposed in evaluation context $c$ and redex $t$, then one may attempt to represent it as a solution of the form $|t \gamma_c|$, where $\gamma_c$ is some CHAM representation of the evaluation context $c$. If this worked, then we could use an airlock operation to isolate that redex from the
rest of the syntactic solution, i.e. \( p \equiv t \gamma \equiv t \gamma \), and thus have it at the same level with the state solution in the configuration solution; this would allow to have rules that match both a syntactic molecule and a state molecule (after an airlock operation is applied on the state solution as well) in the same rule, as needed for the semantics of lookup and assignment. In our example above, we would obtain
\[
\{x = 3 / (x + 2); x \rightarrow 1 \ y \rightarrow 0\} \Rightarrow \{x < (x = 3 / (x + 2); x) \rightarrow 1 \ y \rightarrow 0\}
\]
and the latter could be rewritten with a natural CHAM reaction rule for variable lookup such as
\[
\{x < c\} \{x \rightarrow i < c\} \Rightarrow \{i < c\} \{x \rightarrow i < c\}
\]
Unfortunately, there seems to be no way to achieve such a desirable CHAM representation of syntax. We next attempt and fail to do it in two different ways, and then give an argument why such a representation is actually impossible.

Consider, again, the statement “\(x = 3 / (x + 2)\)”. A naive approach to represent this statement term as a syntactic solution (by means of appropriate heating/cooling rules) is to flatten it into its redex, namely \(x\), and into all its atomic evaluation subcontexts, that is, to represent it as the following solution:
\[
\{x \ (\square + 2) \ (3 / \square) \ (x = \square; )\}
\]
Such a representation can be relatively easily achieved by adding heating/cooling pair rules that correspond to the evaluation strategies (or contexts) of the various language constructs. For example, we can add the following rules corresponding to the evaluation strategies of the assignment and the addition constructs (and two similar ones for the division construct):
\[
\begin{align*}
x &= a; & \Rightarrow & a < \{x = \square; \} \\
(a_1 + a_2) &= a_1 < [\square + a_2] \\
(a_1 + a_2) &= a_2 < [a_1 + \square]
\end{align*}
\]
With such rules, one can now heat or cool syntax as desired, for example:
\[
\begin{align*}
\{x = 3 / (x + 2); \} & \Rightarrow \{(3 / (x + 2)) < \{x = \square; \}\} & \text{(Reaction)} \\
& \Rightarrow \{(3 / (x + 2)) \ (x = \square; )\} & \text{(Airlock)} \\
& \Rightarrow \{x + 2) \ (3 / \square) \ (x = \square; )\} & \text{(Reaction, Chemical, Airlock)} \\
& \Rightarrow \{x \ (\square + 2) \ (3 / \square) \ (x = \square; )\} & \text{(Reaction, Chemical, Airlock)}
\end{align*}
\]
Unfortunately this naive approach is ambiguous, because it cannot distinguish the above from the representation of, say, \(x = (3 / x) + 2;\). The problem here is that the precise structure of the evaluation context is “lost in translation”, so the approach above does not work.

Let us attempt a second approach, namely to guarantee that there is precisely one hole \(\square\) molecule in each syntactic subsolution by using the molecule/solution nesting mechanism available in CHAM. More precisely, let us try to unambiguously represent the statements “\(x = 3 / (x + 2)\)” and “\(x = (3 / x) + 2;\)” as the following two distinct syntactic solutions:
\[
\begin{align*}
\{x \ ((\square + 2) \ ((3 / \square) \ ((x = \square; ))))\} \\
\{x \ ((3 / \square) \ ((\square + 2) \ ((x = \square; ))))\}
\end{align*}
\]
To achieve this, we modify the heating/cooling rules above as follows:
\[
\begin{align*}
(x &= a;) < c & \Rightarrow & a < [(x = \square; ) < c] \\
(a_1 + a_2) &= a_1 < [(\square + a_2) < c] \\
(a_1 + a_2) &= a_2 < [(a_1 + \square) < c]
\end{align*}
\]
With these modified rules, one may now think that one can heat and cool syntax unambiguously:

\[
\begin{align*}
\{ x = 3 / (x + 2) ; \} & \implies \{(x = 3 / (x + 2)) \} \tag{Airlock} \\
& \implies \{(3 / (x + 2)) \} \tag{Reaction} \\
& \implies \{(3 / (x + 2)) \} \tag{Airlock, Membrane} \\
& \implies \{(x + 2) \} \tag{Reaction} \\
& \implies \{(x + 2) \} \tag{Airlock, Membrane} \\
& \implies \{ x \} \tag{All four laws} \\
& \implies \{ x \} \tag{All four laws}
\end{align*}
\]

Unfortunately, the above is not the only way one can heat the solution in question. For example, the following is also a possible derivation, showing that these heating/cooling rules are still problematic:

\[
\begin{align*}
\{ x = 3 / (x + 2) ; \} & \implies \{(x = 3 / (x + 2)) \} \tag{Airlock} \\
& \implies \{(3 / (x + 2)) \} \tag{Reaction} \\
& \implies \{(3 / (x + 2)) \} \tag{Airlock, Membrane} \\
& \implies \{(x + 2) \} \tag{All four laws} \\
& \implies \{ x \} \tag{All four laws}
\end{align*}
\]

Indeed, one can similarly show that

\[
\{ x = (3 / x) + 2 ; \} \implies \{ x \} \tag{hypothesis}
\]

Therefore, this second syntax representation attempt is also ambiguous.

We claim that it is impossible to devise heating/cooling rules in CHAM and representations \( \gamma_c \) of evaluation contexts with the property that

\[
\{ c[t] \} \implies \{ t \} \quad \text{or, equivalently,} \quad \{ c[t] \} \implies \{ t \}
\]

for any term \( t \) and any appropriate evaluation context \( c \). Indeed, if that was possible, then the following derivation could be possible:

\[
\begin{align*}
\{ x = 3 / (x + 2) ; \} & \implies \{(3 / (x + 2)) \} \tag{hypothesis} \\
& \implies \{(x + 2) \} \tag{hypothesis, Chemical} \\
& \implies \{ x \} \tag{hypothesis, Chemical} \\
& \implies \{(3 / x) \} \tag{hypothesis, Chemical} \\
& \implies \{(3 / x) + 2 \} \tag{hypothesis}
\end{align*}
\]

This general impossibility result explains why both our representation attempts above failed, as well as why many other similar attempts are also expected to fail.

The morale of the exercise above is that one should be very careful when using CHAM’s airlock, because in combination with the other CHAM laws it can yield unexpected behaviors. In particular, the Chemical Law makes it impossible to state that a term matches the entire contents of a solution molecule, so one should not rely on the fact that all the remaining contents of a solution is in the membrane following the airlock. In our heating/cooling rules above, for example

\[
\begin{align*}
(x = a; ) & \iff a \in \{(x = a ; ) \} \\
(a_1 + a_2) & \iff a_1 \in \{(a_1 + a_2 ; ) \} \\
(a_1 + a_2) & \iff a_2 \in \{(a_1 + a_2 ; ) \}
\end{align*}
\]

275
our intuition that $c$ matches all the evaluation context solution representation was wrong precisely for that reason. Indeed, it can just as well match a solution representation of a subcontext, which is why we got the unexpected derivation.

**Correct representation of syntax.** We next discuss an approach to representing syntax which is not based on CHAM’s existing solution/membrane mechanism. We borrow from K (see Section 3.12) the idea of flattening syntax in an explicit list of computational tasks. Like in K, we use the symbol $\rightleftharpoons$, read “then” or “followed by”, to separate such computational tasks; to avoid writing parentheses, we here assume that $\rightleftharpoons$ is right-associative and binds less tightly than any other construct. For example, the term “$x = 3 / (x + 2); x = \square$” gets represented as the list term

\[
x \rightleftharpoons \square + 2 \rightleftharpoons 3 / \square \rightleftharpoons x = \square; \rightleftharpoons \square
\]

which reads “process $x$, followed by adding 2 to it, followed by dividing 3 by the result, followed by assigning the obtained result to $x$, which is the final task”. Figure 3.44 shows all the heating/cooling rules that we associate to the various evaluation strategies of the IMP language constructs. These rules allow us to structurally rearrange any well-formed syntactic term so that the next computational task is at the top (left side) of the computation list.

The only rule in Figure 3.44 which does not correspond to the evaluation strategy of some evaluation construct is $s \rightleftharpoons s \rightleftharpoons \square$. Its role is to initiate the decomposition process whenever an unheated statement is detected in the syntax solution. According to CHAM’s laws, these rules can only apply in solutions, so we can derive

\[
\{x = 1; \ x = 3 / (x + 2);\} \overset{\ast}{\rightleftharpoons} \{x = 1; \ \square \ x = 3 / (x + 2); \ \square\}
\]

but there is no way to derive, for example,

\[
\{x = 1; \ x = 3 / (x + 2);\} \overset{\ast}{\rightleftharpoons} \{x = 1; \ (x \rightleftharpoons \square + 2 \rightleftharpoons 3 / \square \rightleftharpoons x = \square; \ \square)\}
\]

The syntactic solution will contain only one syntactic molecule at any given moment, with no subsolutions, which is the reason why the heating/cooling rules in Figure 3.44 that correspond to language construct evaluation strategies need to mention the remaining of the list of computational tasks in the syntactic molecule, $c$, instead of just the interesting part (e.g., $a_1 + a_2 \rightleftharpoons a_1 \rightleftharpoons \square + a_2$, etc.), as we do in K (see

---

Figure 3.44: CHAM heating-cooling rules for IMP.
Figure 3.45: CHAM(IMP): The CHAM of IMP, obtained by adding to the heating/cooling rules in Figure 3.44 the semantic rules for IMP plus the heating rule for state initialization above.

Section 3.12. The heating/cooling rules in Figure 3.44 decompose the syntactic term into any of its possible splits into a redex (the top of the resulting list of computational tasks) and an evaluation context (represented flattened as the rest of the list). In fact, these heating/cooling rules have been almost mechanically derived from the syntax of evaluation contexts for the IMP language constructs in Section 3.7.1 (see Figure 3.30).

Figure 3.45 shows the remaining CHAM rules of IMP, giving the actual semantics of each language construct. Like the heating/cooling rules in Figure 3.44, these rules are also almost mechanically derived from the rules of the reduction semantics with evaluation contexts of IMP in Section 3.7.1 (see Figure 3.31), with the following notable differences:

- Each rule needs to mention the remaining list of computational tasks, c, for the same reason the heating/cooling rules in Figure 3.44 need to mention it (which is explained above).
- There is no equivalent of the characteristic rule of reduction semantics with evaluation contexts. The Membrane Law looks somehow similar, but we cannot take advantage of that because we were not able to use the inherent airlock mechanism of the CHAM to represent syntax (see the failed attempts to represent syntax above).
- The state is organized as a solution using CHAM’s airlock mechanism, instead of just imported as an external data-structure as we did in our previous semantics. We did so because the state data-structure that we used in our previous semantics was a finite-domain partial function (see Section 3.1.2), which was represented as a set of pairs (Section 2.1.2), and it is quite natural to replace any set structures by the inherent CHAM solution mechanism.

Figure 3.46 shows a possible execution of IMP’s CHAM defined above, mentioning at each step which of CHAM’s laws have been applied. Note that the final configuration contains two subsolutions, one for the
We next discuss the CHAM of IMP++

The chemical abstract machine can also define the increment modularly:

\[
\{ \text{int } x, y; \ x = 1; \ x = 3 / (x + 2); \} \rightarrow (\text{Reaction})
\]
\[
\{\{x = 1; \ x = 3 / (x + 2)\}; \ {x, y :: 0}\} \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ x = 3 / (x + 2); \ \rightarrow (\text{Heating, Membrane, Airlock})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Airlock, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Reaction, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
\[
\{\{x = 1; \ \rightarrow (\text{Heating, Membrane})
\]
print(a); ⊕ c ⇐ a ⊕ print(□); ⊕ c

{read() ⊕ c} {input i : w} → {i ⊕ c} {input w}

{print(i); ⊕ c} {output w} → {[]} ⊕ c} {output w : i}

{int x; s} {w} → {s} {x := 0} {input w} {output c}

(CHAM-READ)

(CHAM-PRINT)

(CHAM-PM)

Abrupt Termination

The CHAM semantics of abrupt termination is even more elegant and modular than that using evaluation contexts above, because the other components of the configuration need not be mentioned:

\[ i / 0 ⊕ c → {} \]  \hspace{1cm} (CHAM-DIV-BY-ZERO)

\[ \text{halt}; ⊕ c → {} \]  \hspace{1cm} (CHAM-HALT)

Dynamic Threads

As stated in Section 3.8, the CHAM has been specifically proposed as a model of concurrent computation, based on the chemical metaphor that molecules in solutions can get together and react, with possibly many reactions taking place concurrently. Since there was no concurrency so far in our language, the actual strength of the CHAM has not been seen yet. Recall that the configuration of the existing CHAM semantics of IMP consists of one top-level solution, which contains two subsolutions: a syntactic subsolution holding the remainder of the program organized as a molecule sequentializing computation tasks using the special construct ⊕; and a state subsolution containing binding molecules, each binding a different program variable to a value. As seen in Figures 3.44 and 3.45, most of the CHAM rules involve only the syntactic molecule. The state subsolution is only mentioned when the language construct involves program variables.

The above suggests that all a spawn statement needs to do is to create an additional syntactic subsolution holding the spawned statement, letting the newly created subsolution molecule to float together with the original syntactic molecule in the same top-level solution. Minimalistically, this can be achieved with the following CHAM rule (which does not consider thread termination yet):

\[ \{\text{spawn } s ⊕ c\} → \{[] \rightarrow c\} \{s\} \]

Since the order of molecules in a solution is irrelevant, the newly created syntactic molecule has the same rights as the original molecule in reactions involving the state molecule. We can (correctly) think of each syntactic subsolution as an independently running execution thread. The same CHAM rules we had before (see Figures 3.44 and 3.45) can now also apply to the newly created threads. Moreover, reactions taking place only in the syntactic molecules, which are a majority by a large number, can apply truly concurrently. For example, a thread may execute a loop unrolling step while another thread may concurrently perform an addition. The only restriction regarding concurrency is that rule instances must involve disjoint molecules in order to proceed concurrently. That means that it is also possible for a thread to read or write the state while another thread, truly concurrently, performs a local computation. This degree of concurrency was not possible within the other semantic approaches discussed so far in this chapter.

The rule above only creates threads. It does not collect threads when they complete their computation. One could do that with the simple solution-dissolving rule

\[ \{[]\} → · \]

but the problem is that such a rule cannot distinguish between the original thread and the others, so it would also dissolve the original thread when it completes. This could be considered correct behavior, but, however,
we prefer to distinguish the original thread created statically from the others, which are created dynamically. Specifically, we collect only the terminated threads which were created dynamically. To achieve that, we can flag the newly created threads for collection as below. Here is our complete CHAM semantics of `spawn`:

\[
\textit{Molecule ::= \ldots | \textit{die}}
\]

\[
\{\text{spawn } s \sim c\} \rightarrow \{\emptyset \sim c\} \{s \sim \text{die}\} \tag{CHAM-SPAWN}
\]

\[
\{\emptyset \sim \text{die}\} \rightarrow \cdot \tag{CHAM-DIE}
\]

We conclude this section with a discussion on the concurrency of the CHAM above. As already argued, it allows for truly concurrent computations to take place, provided that their corresponding CHAM rule instances do not overlap. While this already goes far beyond the other semantical approaches in terms of concurrency, it still enforces interleaving wherever it should not.

Consider, for example, a global configuration in which two threads are about to lookup two different variables in the state cell. Even though there are good reasons to allow the two threads to proceed concurrently, the CHAM above will not, because the two rule instances (of the same CHAM lookup rule) overlap on the state molecule. This problem can be ameliorated, to some extent, by changing the structure of the top-level configuration to allow all the variable binding molecules currently in the state subsolution to instead float in the top-level solution at the same level with the threads: this way, each thread can independently grab the binding it is interested in without blocking the state anymore. Unfortunately, this still does not completely solve the true concurrency problem, because one could argue that different threads should also be allowed to concurrently read the same variable. Thus, no matter where the binding of that variable is located, the two rule instances cannot proceed concurrently. Moreover, flattening all the syntactic and the semantic ingredients in a top level solution, as the above “fix” suggests, does not scale. Real-life languages can have many configuration items of various kinds, such as, environments, heaps, function/exception/loop stacks, locks held, and so on. Collapsing the contents of all these items in one flat solution would not only go against the CHAM philosophy, but it would also make it hard to understand and control. The K framework (see Section 3.12) solves this problem by allowing its rules to state which parts of the matched subterm are shared with and which can be concurrently modified by other rules.

**Local Variables**

The simplest approach to adding blocks with local variables to IMP is to follow an idea similar to the one for reduction semantics with evaluation contexts discussed in Section 3.7.2, assuming the procedure presented in Section 3.5.5 for desugaring blocks with local variables into `let` constructs:

\[
\text{let } x = a \text{ in } s \sim c \Leftarrow a \sim \text{let } x = \square \text{ in } s \sim c
\]

\[
\{\text{let } x = i \text{ in } s \sim c\} \{\sigma\} \rightarrow \{s x = \sigma(x); c\} \{\sigma[i/x]\} \tag{CHAM-LET}
\]

As it was the case in Section 3.7.2, the above is going to be problematic when we add `spawn` to the language, too. However, recall that in this language experiment we pretend each language extension is final, in order to understand the modularity and flexibility to change of each semantic approach.

The approach above is very syntactic in nature, following the intuitions of evaluation contexts. In some sense, the above worked because we happened to have an assignment statement in our language, which we used for recovering the value of the bound variable. Note, however, that several computational steps were wasted because of the syntactic translations. What we would have really liked to say is “let \textit{Id} = \textit{Int} in \textit{Context} is a special evaluation context where the current state is updated with the binding
whenever \texttt{let} is passed through top-down, and where the state is recovered whenever \texttt{let} is passed through bottom-up”. This was not possible to say with evaluation contexts. When using the CHAM, we are free to disobey the syntax. For example, the alternative definition below captures the essence of the problem and wastes no steps (although it is still problematic when combined with \texttt{spawn}):

\[
\texttt{let } x = a \texttt{ in } s \sim c \vdash a \sim \texttt{let } x = \boxtimes \texttt{ in } s \sim c \\
\{\texttt{let } x = i \texttt{ in } s \sim c \} \{\sigma\} \rightarrow \{s \sim \texttt{let } x = \sigma(x) \texttt{ in } \boxtimes \sim c\} \{\sigma[i/x]\} \\
\{\} \sim \texttt{let } x = v \texttt{ in } \boxtimes \sim c \{\sigma\} \rightarrow \{\} \sim c \{\sigma[v/x]\}
\]

In words, the \texttt{let} is first heated/\texttt{cooled} in the binding expression. Once that becomes an integer, the \texttt{let} is then only heated in its body statement, at the same time updating the state molecule with the binding and storing the return value in the residual \texttt{let} construct. Once the \texttt{let} body statement becomes {}, the solution is cooled down by discarding the residual \texttt{let} construct and recovering the state appropriately (we used a “value” \(v\) instead of an integer \(i\) in the latter rule to indicate the fact that \(v\) can also be \(\bot\)).

Of course, the substitution-based approach discussed in detail in Sections 3.5.6 and 3.7.2 can also be adopted here if one is willing to pay the price for using it:

\[
\texttt{let } x = a \texttt{ in } s \sim c \vdash a \sim \texttt{let } x = \boxtimes \texttt{ in } s \sim c \\
\{\texttt{let } x = i \texttt{ in } s \sim c \} \{\sigma\} \rightarrow \{s[x'/x] \sim c\} \{\sigma[i/x']\} \text{ if } x' \text{ is a fresh variable}
\]

Putting Them All Together

Putting together all the language features defined in CHAM above is a bit simpler and more modular than in MSOS (see Section 3.6.2): all we have to do is to take the union of all the syntax and semantics of all the features, removing the original rule for the initialization of the solution (that rule was already removed as part of the addition of input/output to IMP); in MSOS, we also had to add the halting attribute to the labels, which we do not have to do in the case of the CHAM.

Unfortunately, like in the case of the reduction semantics with evaluation contexts of IMP++ in Section 3.7.2, the resulting language is flawed. Indeed, a thread spawned from inside a \texttt{let} would be created its own molecule in the top-level solution, which would execute concurrently with all the other execution threads, including its parent. Thus, there is the possibility that the parent will advance to the assignment recovering the value of the \texttt{let}-bound variable before the spawned thread terminates, in which case the bound variable would be changed in the state by the parent thread, “unexpectedly” for the spawned thread. One way to address this problem is to rename the bound variable into a fresh variable within the \texttt{let} body statement, like we did above, using a substitution operation. Another is to split the state into an environment mapping variables to locations and a store mapping locations to values, and to have each thread consist of a solution holding both its code and its environment. Both these solutions were also suggested in Section 3.5.6 when we discussed how to make small-step SOS correctly capture all the behaviors of the resulting IMP++.

3.8.3 CHAM in Rewrite Logic

As explained above, CHAM rewriting cannot be immediately captured as ordinary rewriting modulo solution multiset axioms such as associativity, commutativity and identity. The distinction between the two arises essentially when a CHAM rule involves only one top-level molecule which is not a solution, because the CHAM laws restrict the applications of such a rule only in solutions while ordinary rewriting allows such rules to apply everywhere. To solve this problem, we wrap each rule in a solution context, that is, we translate CHAM rules of the form

\[
m_1 \ m_2 \ \ldots \ m_k \rightarrow m_1' \ m_2' \ \ldots \ m_l'
\]

281
into corresponding rewrite logic rules of the form
\[
\{m_1, m_2, \ldots, m_k, Ms\} \rightarrow \{m'_1, m'_2, \ldots, m'_l, Ms\}
\]
where the only difference between the original CHAM terms \(m_1, m_2, \ldots, m_k, m'_1, m'_2, \ldots, m'_l\) and their algebraic variants \(\overline{m_1}, \overline{m_2}, \ldots, \overline{m_k}, \overline{m'_1}, \overline{m'_2}, \ldots, \overline{m'_l}\) is that the meta-variables appearing in the former (recall that CHAM rules are schemata) are turned into variables of corresponding sorts in the latter, and where \(Ms\) is a variable of sort \(\text{Bag}\{\text{Molecule}\}\) that does not appear anywhere else in \(\overline{m_1}, \overline{m_2}, \ldots, \overline{m_k}, \overline{m'_1}, \overline{m'_2}, \ldots, \overline{m'_l}\).

With this representation of CHAM rules into rewrite logic rules, it is obvious that rewrite logic’s rewriting captures both the Reaction Law and the Chemical Law of the CHAM. What is less obvious is that it also captures the Membrane Law. Indeed, note that the Membrane Law allows rewrites to take place only when the global term is a solution, while rewriting logic allows rewrites to take place anywhere. However, the rewrite logic rule representation above generates only rules that rewrite solutions into solutions. Thus, if the original term to rewrite is a solution, then so will it stay during the entire rewriting process, and so the Membrane Law is also naturally captured by rewrite logic derivations. However, if the original term to rewrite is a proper molecule (which is not a solution), then so will it stay during the entire rewrite logic’s rewriting while the CHAM will not reduce it at all. Still, it is important to understand that in this case the corresponding rewrite logic theory can perform rewrite steps (in subsolutions of the original term) which are not possible under the CHAM, in particular that it may lead to non-termination in situations where the CHAM is essentially stuck. To reconcile this inherent difference between the CHAM and rewrite logic, we make the reasonable assumption that the original terms to rewrite can only be solutions. Note that the CHAM sequents in Definition 23 already assume that one only derives solution terms.

The only CHAM law which has not been addressed above is the Airlock Law. Rewrite logic has no built-in construct resembling CHAM’s airlock, but its multiset matching is powerful enough to allow us to capture the airlock’s behavior through rewrite rules. One possibility is to regard the airlock operation like any other molecular construct. Indeed, from a rewrite logic perspective, the Airlock Law says that any molecule inside a solution can be matched and put into an airlock next to the remaining solution wrapped into a membrane, and this process is reversible. This behavior can be achieved through the following two (opposite) rewrite logic rules, where \(M\) is a molecule variable and \(Ms\) is a bag-of-molecules variable:
\[
\begin{align*}
\{M, Ms\} & \rightarrow \{M \triangleright \{Ms\}\} \\
\{M \triangleright \{Ms\}\} & \rightarrow \{M, Ms\}
\end{align*}
\]

Another possibility to capture airlock’s behavior in rewrite logic is to attempt to eliminate it completely and replace it with matching modulo multiset axioms. While this appears to be possible in many concrete situations, we are however not aware of any general solution to do so systematically for any CHAM. The question is whether the elimination of the airlock is indeed safe, in the sense that the resulting rewrite logic theory does not lose any of the original CHAM’s behaviors. One may think that thanks to the restricted form of CHAM’s rules, the answer is immediately positive. Indeed, since the CHAM disallows any other constructs for solutions except its built-in membrane operation (a CHAM can only add new syntactic constructs for molecules, but not for solutions) and since solution subterms can either contain only one molecule or otherwise be meta-variables (to avoid multiset matching), we can conclude that in any CHAM rule, a subterm containing an airlock operation at its top can only be of the form \(m \triangleright \{m'\}\) or of the form \(m \triangleright s\) with \(s\) a meta-variable. Both these cases can be uniformly captured as subterms of the form \(\overline{m} \triangleright \{ms\}\) with \(ms\) a term of sort \(\text{Bag}\{\text{Molecule}\}\) in rewrite logic, the former by taking \(k = 1\) and \(ms = m'\) and the latter by replacing the metavariable \(s\) with a term of the form \(\{Ms\}\) everywhere in the rule, where \(Ms\) is a fresh variable of sort \(\text{Bag}\{\text{Molecule}\}\). Unfortunately, it is not clear how we can eliminate the airlock from subterms of the form
sorts:
- Molecule, Solution, Bag{Molecule}

subsorts:
- Solution < Molecule
  // One may also need to subsort to Molecule specific syntactic categories (Int, Bool, etc.)

operations:
- \{ \} : Bag{Molecule} → Solution // membrane operator
- \{ \} → \_ : Molecule × Solution → Molecule // airlock operator
  // One may also need to define specific syntactic constructs for Molecule (_ + _, _ ↦ _, etc.)

rules:
- // Add the following two generic (i.e., same for all CHAMs) airlock rewrite logic rules:
  \{ M Ms \} ↔ \{ M ⊉ \{Ms\} \} // M, Ms variables of sorts Molecule, Bag{Molecule}, resp.
  // For each specific CHAM rule \( m_1 m_2 \ldots m_k \rightarrow m'_1 m'_2 \ldots m'_l \) add a rewrite logic rule
  \{ \overline{m}_1 \overline{m}_2 \ldots \overline{m}_k Ms \} → \{ \overline{m'}_1 \overline{m'}_2 \ldots \overline{m'}_l Ms \} // Ms variable of sort Bag{Molecule}
  // where \( \overline{m} \) replaces each meta-variable in \( m \) by a variable of corresponding sort.

Figure 3.47: Embedding of a chemical abstract machine into rewrite logic (CHAM ↔ \( R_{CHAM} \)).

\( \overline{m} \triangleright \{ ms \} \). If such terms appear in a solution or at the top of the rule, then one can replace them by their corresponding bag-of-molecule term \( \overline{m} ms \). However, if they appear in a proper molecule context (i.e., not in a solution context), then they cannot be replaced by \( \overline{m} ms \) (first, we would get a parsing error by placing a bag-of-molecule term in a molecule place; second, reactions cannot take place in \( ms \) anymore, because it is not surrounded by a membrane). We cannot replace \( \overline{m} \triangleright \{ ms \} \) by \( \overline{m} \{ ms \} \) either, because that would cause a double membrane when the thus modified airlock reaches a solution context. Therefore, we keep the airlock.

Putting all the above together, we can associate a rewrite logic theory to any CHAM as shown in Figure 3.47 for simplicity, we assumed that the CHAM has only one Molecule syntactic category and, implicitly, only one corresponding Solution syntactic category. In Figure 3.47 and elsewhere in this section, we use the notation \( left \leftrightarrow right \) as a shorthand for two opposite rewrite rules, namely for both \( left \rightarrow right \) and \( right \rightarrow left \). The discussion above implies the following result:

Theorem 21. (Embedding of the chemical abstract machine into rewrite logic) If CHAM is a chemical abstract machine, sol and sol’ are two solutions, and \( R_{CHAM} \) is the rewrite logic theory associated to CHAM as in Figure 3.47, then the following hold:

1. CHAM ⊢ sol → sol’ if and only if \( R_{CHAM} \triangleright \overline{sol} \rightarrow^1 \overline{sol'} \);
2. CHAM ⊢ sol →* sol’ if and only if \( R_{CHAM} \triangleright \overline{sol} \rightarrow \overline{sol'} \).

Therefore, one-step solution rewriting in \( R_{CHAM} \) corresponds precisely to one-step solution rewriting in the original CHAM and thus, one can use \( R_{CHAM} \) as a replacement for CHAM for any reduction purpose. Unfortunately, this translation of CHAM into rewrite logic does not allow us to borrow the latter’s concurrency to obtain the desired concurrent rewriting computational mechanism of the former. Indeed, the desired CHAM concurrency says that “different rule instances can apply concurrently in the same solution as far as they act on different molecules”. Unfortunately, the corresponding rewrite logic rule instances cannot apply concurrently according to rewrite logic’s semantics because both instances would match the entire solution, including the membrane (and rule instances which overlap cannot proceed concurrently in rewrite logic—see Section 2.5).
The CHAM of IMP in Rewrite Logic

Figure 3.48 shows the rewrite theory $R_{\text{CHAM(IMP)}}$ obtained by applying the generic transformation procedure in Figure 3.47 to the CHAM of IMP discussed in this section and summarized in Figures 3.44 and 3.45. In addition to the generic CHAM syntax, as indicated in the comment under subsorts in Figure 3.47, we also subsort the built-in sorts of IMP, namely Int, Bool, and Id, to Molecule. The “followed by” $\rightarrow$ construct for molecules is necessary for defining the evaluation strategies of the various IMP language constructs as explained above; we believe that some similar operator is necessary when defining any programming language whose constructs have evaluation strategies because, as explained above, it appears that the CHAM airlock operator is not suitable for this task. For notational simplicity, we stick to our previous convention that $\rightarrow$ is right associative and binds less tight than any other molecular construct. Finally, we add molecular constructs corresponding to all the syntax that we need in order to define the IMP semantics, which includes syntax for language constructs, for evaluation contexts, and for the state and state initialization.

The rewrite rules in Figure 3.48 are straightforward, following the transformation described in Figure 3.47. We grouped them in four categories: (1) the airlock rule is reversible and precisely captures the CHAM Airlock Law; (2) the heating/cooling rules, also reversible, capture the evaluation strategies of IMP’s language constructs (see Figure 3.44); (3) the semantic rules are irreversible and capture the computational steps of the IMP semantics (see Figure 3.45); (4) the state initialization rule corresponds to the heating rule in Figure 3.45.

Our CHAM-based rewrite logic semantics of IMP in Figure 3.48 follows blindly the CHAM of IMP in Figures 3.44 and 3.45. All it does is to mechanically apply the transformation in Figure 3.47 without making any attempts to optimize the resulting rewrite logic theory. For example, it is easy to see that the syntactic molecules will always contain only one molecule. Also, it is easy to see that the state molecule can be initialized in such a way that at any moment during the initialization rewriting sequence any subsolution containing a molecule of the form $xl \mapsto i$, with xl a proper list, will contain no other molecule. Finally, one can also notice that the top level solution will always contain only two molecules, namely what we called a syntactic solution and a state solution. All these observations suggest that we can optimize the rewrite logic theory in Figure 3.48 by deleting the variable $Ms$ from every rule except the airlock ones. While one can do that for our simple IMP language here, one has to be careful with such optimizations in general. For example, we later on add threads to IMP (see Section 3.5), which implies that the top level solution will contain a dynamic number of syntactic subsolutions, one per thread; if we remove the $Ms$ variable from the lookup and assignment rules in Figure 3.48, then those rules will not work whenever there are more than two threads running concurrently. On the other hand, the $Ms$ variable in the rule for variable declarations can still be eliminated. The point is that one needs to exercise care when one attempts to hand-optimize the rewrite logic theories resulting from mechanical semantic translations.

★ The CHAM of IMP in Maude

Like for the previous semantics, it is relatively straightforward to mechanically translate the corresponding rewrite theories into Maude modules. However, unlike in the previous semantics, the resulting Maude modules are not immediately executable. The main problem is that, in spite of its elegant chemical metaphor, the CHAM was not conceived to be blindly executable. For example, most of the heating and cooling rules tend to be reversible, leading to non-termination of the underlying rewrite relation. Non-termination is not a problem per se in rewrite logic and in Maude, because one can still use other formal analysis capabilities of these such as search and model checking, but from a purely pragmatic perspective it is rather inconvenient not to be able to execute an operational semantics of a language, particularly of a simple one like our IMP. Moreover, since the state space of a CHAM can very quickly grow to unmanageable sizes even when the
operations:

\[ \text{Solution} \rightarrow \text{Solution} \]  // generic membrane

\[ \text{Molecule} \times \text{Solution} \rightarrow \text{Molecule} \]  // generic airlock

\[ \text{Molecule} \times \text{Molecule} \rightarrow \text{Molecule} \]  // “followed by” operator, for evaluation strategies

// Plus all the IMP language constructs and evaluation contexts (all listed in Figure 3.30)
// collapsing all syntactic categories different from \( \text{Int}, \text{Bool} \) and \( \text{Id} \) into \( \text{Molecule} \)

rules:

// Airlock:
\[ \{ M \cap M_s \} \rightarrow \{ M \cap \{ M \cap M_s \} \} \]

// Heating/cooling rules corresponding to the evaluation strategies of IMP’s constructs:
\[ \{ A_1 + A_2 \rightarrow C \} M_s \rightarrow \{ (A_1 \cap \Delta) + (A_2 \cap C) \} M_s \]
\[ \{ A_1 + A_2 \rightarrow C \} M_s \rightarrow \{ (A_2 \cap A_1 + \square \cap C) \} M_s \]
\[ \{ A_1 / A_2 \rightarrow C \} M_s \rightarrow \{ (A_1 \cap \square) / A_2 \cap C) \} M_s \]
\[ \{ A_1 / A_2 \rightarrow C \} M_s \rightarrow \{ (A_2 \cap A_1 / \square \cap C) \} M_s \]
\[ \{ A_1 <= A_2 \rightarrow C \} M_s \rightarrow \{ (A_1 \cap \square <= A_2 \cap C) \} M_s \]
\[ \{ I_1 <= A_2 \rightarrow C \} M_s \rightarrow \{ (A_2 \cap I_1 <= \square \cap C) \} M_s \]
\[ \{ (B \rightarrow C) M_s \rightarrow \{ (B \rightarrow \square \rightarrow C) M_s \} \]
\[ \{ B_1 \& B_2 \rightarrow C \} M_s \rightarrow \{ B_1 \rightarrow \square \& B_2 \rightarrow C \} M_s \]
\[ \{ X \rightarrow A \rightarrow C \} M_s \rightarrow \{ (A = \square) \rightarrow C \} M_s \]
\[ \{ S_1 \rightarrow S_2 \rightarrow C \} M_s \rightarrow \{ S_1 \rightarrow \square \rightarrow S_2 \rightarrow C \} M_s \]
\[ \{ S \rightarrow M_s \} \rightarrow \{ (S \rightarrow \square) M_s \} \]
\[ \{ (\text{if} (B) S_1 \text{else} S_2 \rightarrow C) M_s \} \rightarrow \{ (B \rightarrow \text{if} (\square) S_1 \text{else} S_2 \rightarrow C) M_s \} \]

// Semantic rewrite rules corresponding to reaction computational steps
\[ \{ X \rightarrow C \} \{ X \rightarrow I \rightarrow \sigma \} M_s \rightarrow \{ (I \rightarrow C \rightarrow I \rightarrow \sigma) M_s \} \]
\[ \{ I_1 + I_2 \rightarrow C \} M_s \rightarrow \{ (I_1 + I_2 \rightarrow C) M_s \} \]
\[ \{ I_1 / I_2 \rightarrow C \} M_s \rightarrow \{ (I_1 / I_2 \rightarrow C) M_s \} \quad \text{if} \quad I_2 \neq 0 \]
\[ \{ I_1 <= I_2 \rightarrow C \} M_s \rightarrow \{ (I_1 \leq I_2 \rightarrow C) M_s \} \]
\[ \{ (\text{true} \rightarrow C) M_s \rightarrow \{ (\text{false} \rightarrow C) M_s \} \]
\[ \{ (\text{false} \rightarrow C) M_s \rightarrow \{ (\text{false} \rightarrow C) M_s \} \]
\[ \{ (\text{true} \& B_2 \rightarrow C) M_s \rightarrow \{ (B_2 \rightarrow C) M_s \} \]
\[ \{ (\text{false} \& B_2 \rightarrow C) M_s \rightarrow \{ (\text{false} \& B_2 \rightarrow C) M_s \} \]
\[ \{ S \rightarrow C \} M_s \rightarrow \{ (S \rightarrow C) M_s \} \]
\[ \{ X = I ; \rightarrow C \} \{ X \rightarrow I \rightarrow \sigma \} M_s \rightarrow \{ (X \rightarrow I \rightarrow \sigma) M_s \} \]
\[ \{ (S_2 \rightarrow C) M_s \rightarrow \{ (S_2 \rightarrow C) M_s \} \]
\[ \{ (\text{if} (\text{true}) S_1 \text{else} S_2 \rightarrow C) M_s \rightarrow \{ (S_1 \rightarrow C) M_s \} \]
\[ \{ (\text{false}) S_1 \text{else} S_2 \rightarrow C) M_s \rightarrow \{ (S_2 \rightarrow C) M_s \} \]
\[ \{ (\text{while} (B) S \rightarrow C) M_s \rightarrow \{ (\text{if} (B) S \text{while} (B) S \text{else} []) \rightarrow C) M_s \} \]
\[ \{ (\text{int} x l ; s) M_s \rightarrow \{ (s) \rightarrow x l \rightarrow 0) M_s \} \]

// State initialization:
\[ \{ X, I \rightarrow I \} M_s \rightarrow \{ (X \rightarrow I \rightarrow (X I \rightarrow I)) M_s \} \]

Figure 3.48: \( \mathcal{R}_{\text{CHAM}(\text{IMP})} \): The CHAM of IMP in rewrite logic.
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op _#_ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  var M : Molecule . var Ms : Bag{Molecule} .
  rl {| M # Ms |} => {| M <| {| Ms |} |} .
  rl {| M <| {| Ms |} |} => {| M # Ms |} .
endm

mod IMP-CHAM-SYNTAX is including PL-INT + CHAM .
  subsort Int < Molecule .
  --- Define all the IMP constructs as molecule constructs
  op _+_ : Molecule Molecule -> Molecule [prec 33 gather (E e) format (d b o d)] .
  op _/_ : Molecule Molecule -> Molecule [prec 31 gather (E e) format (d b o d)] .
  --- ... and so on
  --- Add the hole as basic molecular construct, to allow for building contexts as molecules
  op [] : -> Molecule .
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-1 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule . var Ms : Bag{Molecule} .
  --- + strict in its first argument
  rl {| (A1 + A2) # Ms |} => {| (A1 <| {| [] + A2 |}) # Ms |} .
  rl {| (A1 <| {| [] + A2 |}) # Ms |} => {| (A1 + A2) # Ms |} .
  --- / strict in its second argument
  rl {| (A1 / A2) # Ms |} => {| (A2 <| {| A1 / [] |}) # Ms |} .
  rl {| (A2 <| {| A1 / [] |}) # Ms |} => {| (A1 / A2) # Ms |} .
  --- ... and so on
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-2 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule . var Ms : Bag{Molecule} . var C : Solution .
  --- + strict in its first argument
  rl {| (A1 + A2 <| C) # Ms |} => {| (A1 <| {| [] + A2 <| C |}) # Ms |} .
  rl {| (A1 <| {| [] + A2 <| C |}) # Ms |} => {| (A1 + A2 <| C) # Ms |} .
  --- / strict in its second argument
  rl {| (A1 / A2 <| C) # Ms |} => {| (A2 <| {| A1 / [] <| C |}) # Ms |} .
  rl {| (A2 <| {| A1 / [] <| C |}) # Ms |} => {| (A1 / A2 <| C) # Ms |} .
  --- ... and so on
endm

Figure 3.49: Failed attempts to represent the CHAM of IMP in Maude using the airlock mechanism to define evaluation strategies. This figure also highlights the inconvenience of redefining IMP’s syntax (the module IMP-CHAM-SYNTAX needs to redefine the IMP syntax as molecule constructs).
state space of the represented program is quite small, a direct representation of a CHAM in Maude can easily end up having only a theoretical relevance.

Before addressing the non-termination issue, it is instructive to discuss how a tool like Maude can help us pinpoint and highlight potential problems in our definitions. For example, we have previously seen how we failed, in two different ways, to use CHAM’s airlock operator to define the evaluation strategies of the various IMP language constructs. We have noticed those problems with using the airlock for evaluation strategies by actually experimenting with CHAM definitions in Maude, more precisely by using Maude’s search command to explore different behaviors of a program. We next discuss how one can use Maude to find out that both our attempts to use airlock for evaluation strategies fail. Figure 3.49 shows all the needed modules. CHAM defines the generic syntax of the chemical abstract machine together with its airlock rules in Maude, assuming only one type of molecule. IMP-CHAM-SYNTAX defines the syntax of IMP as well as the syntax of IMP’s evaluation contexts as a syntax for molecules; since there is only one syntactic category for syntax now, namely Molecule, adding □ as a Molecule constant allows for Molecule to also include all the IMP evaluation contexts, as well as many other garbage terms (e.g., □ + □, etc.). The module IMP-HEATING-COOLING-CHAM-FAILED-1 represents in Maude, using the general translation of CHAM rules into rewrite logic rules shown in Figure 3.47 heating/cooling rules of the form

\[ a_1 + a_2 \Rightarrow a_1 \triangleleft [\square + a_2] \]

Only two such groups of rules are shown, which is enough to show that we have a problem. Indeed, all four Maude search commands below succeed:

\[
\text{search}\[1\] \{ | 3 / (1 + 2) | \} \Rightarrow* \{ | 1 | (\square + 2) | 3 / [\square] | \} . \\
\text{search}\[1\] \{ | 1 | (\square + 2) | 3 / [\square] | \} \Rightarrow* \{ | 3 / (1 + 2) | \} . \\
\text{search}\[1\] \{ | (3 / 1) + 2 | \} \Rightarrow* \{ | 1 | (\square + 2) | 3 / [\square] | \} . \\
\text{search}\[1\] \{ | 1 | (\square + 2) | 3 / [\square] | \} \Rightarrow* \{ | (3 / 1) + 2 | \} . 
\]

That means that the Maude solution terms \{ | 3 / (1 + 2) | \} and \{ | (3 / 1) + 2 | \} can rewrite into each other, which is clearly wrong. Similarly, IMP-HEATING-COOLING-CHAM-FAILED-2 represents in Maude heating/cooling rules of the form

\[ (a_1 + a_2) \triangleleft c \Rightarrow a_1 \triangleleft [(\square + a_2) \triangleleft c] \]

One can now check that this second approach gives us what we wanted, that is, a chemical representation of syntax where each subsolution directly contains no more than one □:

\[
\text{search}\[1\] \{ | 3 / (1 + 2) | \} \Rightarrow* \{ | 1 | (\square + 2) | 3 / [\square] | \} . \\
\text{search}\[1\] \{ | (3 / 1) + 2 | \} \Rightarrow* \{ | 1 | (3 / [\square]) | \square + 2 | [\square] | \} .
\]

Indeed, both Maude search commands above succeed. Unfortunately, the following commands

\[
\text{search}\[1\] \{ | 3 / (1 + 2) | \} \Rightarrow* \{ | 1 | (\square + 2) | 3 / [\square] | \} . \\
\text{search}\[1\] \{ | (3 / 1) + 2 | \} \Rightarrow* \{ | 1 | (\square + 2) | 3 / [\square] | \} . \\
\text{search}\[1\] \{ | 1 | (\square + 2) | 3 / [\square] | \} \Rightarrow* \{ | (3 / 1) + 2 | \} .
\]

also succeed, showing that our second attempt to use the airlock for evaluation strategies fails, too. One could also try the following, which should also succeed (proof is the searches above):

\[
\text{search}\[1\] \{ | 3 / (1 + 2) | \} \Rightarrow* \{ | 3 / (1 + 2) | \} . \\
\text{search}\[1\] \{ | (3 / 1) + 2 | \} \Rightarrow* \{ | 1 | (3 / 1) + 2 | \} .
\]
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op _#_ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  op {|_} : Bag{Molecule} -> Solution .
  op _<_|_ : Molecule Solution -> Molecule [prec 110] .
--- The airlock is unnecessary in this particular example.
--- We keep it, though, just in case will be needed as we extend the language.
--- We comment the two airlock rules below out to avoid non-termination.
--- Otherwise we would have to use slower search commands instead of rewrite.
--- var M : Molecule . var Ms : Bag{Molecule} .
--- rl {| M # Ms |} => {| M <| {| Ms |} |} .
--- rl {| M <| {| Ms |} |} => {| M # Ms |} .
endm

Figure 3.50: Generic representation of the CHAM in Maude.

However, on our machine (Linux, 2.4GHz, 8GB memory) Maude 2 ran out of memory after several minutes when asked to execute any of the two search commands above.

Figures 3.50, 3.51 and 3.52 give a correct Maude representation of CHAM(IMP), based on the rewrite logic theory \( R_{CHAM(IMP)} \) in Figure 3.48. Three important observations were the guiding factors of our Maude semantics of CHAM(IMP):

1. While the approach to syntax in Figure 3.49 elegantly allows to include the syntax of evaluation contexts into the syntax of molecules by simply defining \( \Box \) as a molecular construct, unfortunately it still requires us to redefine the entire syntax of IMP into specific constructs for molecules. This is inconvenient at best. We can do better by simply subsorting to Molecule all those syntactic categories that the approach in Figure 3.49 would collapse into Molecule. This way, any fragment of IMP code parses to a subsort of Molecule, so in particular to Molecule. Unfortunately, evaluation contexts are now ill-formed; for example, \( \Box + 2 \) attempts to sum a molecule with an integer, which does not parse. To fix this, we define a new sort for the \( \Box \) constant, say Hole, and declare it as a subsort of each IMP syntactic category that is subsorted to Molecule. In particular, Hole is a subsort of AExp, so \( \Box + 2 \) parses to AExp.

2. As discussed above, most of CHAM’s heating/cooling rules are reversible and thus, when regarded as rewrite rules, lead to non-termination. To avoid non-termination, we restrict the heating/cooling rules so that heating only applies when the heated subterm has computational contents (i.e., it is not a result) while cooling only applies when the cooled term is completely processed (i.e., it is a result). This way, the heating and the cooling rules are applied in complementary situations, in particular they are not reversible anymore, thus avoiding non-termination. To achieve this, we introduce a subsort Result of Molecule, together with subsorts of it corresponding to each syntactic category of IMP as well as with an explicit declaration of IMP’s results as appropriate terms of corresponding result sort.

3. The CHAM’s philosophy is to represent data, in particular program states, using its builtin support for solutions as multisets of molecules, and to use airlock operations to extract pieces of data from such solutions whenever needed. This philosophy is justified both by chemical and by mathematical intuitions, namely that one needs an additional step to observe inside a solution and, respectively, that multiset matching is a complex operation (it is actually an intractable problem) whose complexity
mod IMP-HEATING-COOLING-CHAM is including IMP-SYNTAX + CHAM.
sorts Hole ResultAExp ResultBExp ResultStmt Result.
subsorts Hole < AExp BExp Stmt < Molecule.
subsorts ResultAExp ResultBExp ResultStmt < Result < Molecule.
subsorts Int < ResultAExp < AExp.
subsorts Bool < ResultBExp < BExp.
subsorts ResultStmt < Block.
op {} : -> ResultStmt [ditto].
op _=#_ : Molecule Molecule -> Molecule [gather(e E) prec 120].
var X : Id . var C : [Molecule] . var A A1 A2 : AExp . var R R1 R2 : Result.
var B B1 B2 : BExp . var I I1 I2 : Int . var S S1 S2 : Stmt . var Ms : Bag{Molecule}.
crl {| (A1 + A2 # C) # Ms |} => {| (A1 "# [ ] + A2 "# C) # Ms |} if notBool(A1 :: Result).
rl {| (R1 "# [ ] + A2 "# C) # Ms |} => {| (R1 + A2 "# C) # Ms |}.

crl {| (A1 + A2 "# C) # Ms |} => {| (A2 "# A1 + [ ] "# C) # Ms |} if notBool(A2 :: Result).
rl {| (R2 "# A1 + [ ] "# C) # Ms |} => {| (A1 + R2 "# C) # Ms |}.

crl {| (A1 / A2 "# C) # Ms |} => {| (A1 "# A1 / [ ] "# C) # Ms |} if notBool(A1 :: Result).
rl {| (R1 "# A1 / [ ] "# C) # Ms |} => {| (A1 / R1 "# C) # Ms |}.

crl {| (A1 <= A2 "# C) # Ms |} => {| (A1 "# [ ] <= A2 "# C) # Ms |} if notBool(A2 :: Result).
rl {| (R1 "# [ ] <= A2 "# C) # Ms |} => {| (A1 <= R1 "# C) # Ms |}.

crl {| (! B "# C) # Ms |} => {| (B "# ! [ ] "# C) # Ms |} if notBool(B :: Result).
rl {| (R "# ! [ ] "# C) # Ms |} => {| (! R "# C) # Ms |}.

crl {| (B1 && B2 "# C) # Ms |} => {| (B1 "# [ ] && B2 "# C) # Ms |} if notBool(B1 :: Result).
rl {| (R1 "# [ ] && B2 "# C) # Ms |} => {| (B1 && R1 "# C) # Ms |}.

crl {| (X = A ; "# C) # Ms |} => {| (A "# X = [ ] ; "# C) # Ms |} if notBool(A :: Result).
rl {| (R "# X = [ ] ; "# C) # Ms |} => {| (X = R ; "# C) # Ms |}.

crl {| (S1 S2 "# C) # Ms |} => {| (S1 "# S2 "# C) # Ms |} if notBool(S1 :: Result).
rl {| (R1 "# S2 "# C) # Ms |} => {| (R1 S2 "# C) # Ms |}.

crl {| S # Ms |} => {| (S "# [ ]) # Ms |} if notBool(S :: Result).
rl {| (R "# [ ]) # Ms |} => {| (R # Ms |}.

crl {| (if (B) S1 else S2 "# C) # Ms |} => {| (B "# if ([ ]) S1 else S2 "# C) # Ms |} if notBool(B :: Result).
rl {| (R "# if ([ ]) S1 else S2 "# C) # Ms |} => {| (if (R) S1 else S2 "# C) # Ms |}.
endm

Figure 3.51: Efficient heating-cooling rules for IMP in Maude.

289
mod IMP-SEMANTICS-CHAM is including IMP-HEATING-COOLING-CHAM + STATE .
subsort Pgm State < Molecule .
var X : Id . var Xl : List(Id) . var C : Molecule . var Ms : Bag{Molecule} .
var Sigma : State . var B B2 : BExp . var I J I1 I2 : Int . var S S1 S2 : Stmt .
rl { | { X > C } | # | { X |-> I & Sigma } | # Ms | } => { | { I > C } | # | { X |-> I & Sigma } | # Ms | } .
rl { | (I1 + I2 > C) # Ms | } => { | (I1 +Int I2 > C) # Ms | } .
crl { | (I1 / I2 > C) # Ms | } => { | (false > C) # Ms | } if I2 /=Int Bool 0 .
rl { | (I1 <= I2 > C) # Ms | } => { | (I1 <=Int I2 > C) # Ms | } .
rl { | (! true > C) # Ms | } => { | (true > C) # Ms | } .
rl { | (true && B2 > C) # Ms | } => { | (B2 > C) # Ms | } .
rl { | (false && B2 > C) # Ms | } => { | (false > C) # Ms | } .
rl { | (S) > C # Ms | } => { | (S > C) # Ms | } .
rl { | { X = I ; > C |} # | { X |-> J & Sigma } | # Ms | } => { | { X > C |} # | { X |-> I & Sigma } | # Ms | } .
rl { | (if (true) S1 else S2 > C) # Ms | } => { | (if (true) S1 else S2 > C) # Ms | } .
rl { | (if (false) S1 else S2 > C) # Ms | } => { | (if (false) S1 else S2 > C) # Ms | } .
rl { | (while (B) S > C) # Ms | } => { | (while (B) S < C else ) > C) # Ms | } .
rl { | (int Xl ; S) # Ms | } => { | S # | { Xl |-> 0 |} # Ms | } .
endm

Figure 3.52: The CHAM of IMP in Maude.

cannot be simply “swept under the carpet”. While we agree with these justifications for the airlock
operator, one should also note that impressive progress has been made in the last two decades, after
the proposal of the chemical abstract machine, in terms of multiset matching. For example, languages like
Maude build upon very well-engineered multiset matching techniques. We believe that these recent
developments justify us, at least in practical language definitions, to replace the expensive airlock
operation of the CHAM with the more available and efficient multiset matching of Maude.

Figure 3.50 shows the generic CHAM syntax that we extend in order to define CHAM(IMP). Since we use
Maude’s multiset matching instead of state airlock and since we cannot use the airlock for evaluation strategies
either, there is effectively no need for the airlock in our Maude definition of CHAM(IMP). Moreover, since
the airlock rules are reversible, their introduction would yield non-termination. Consequently, we have plenty
of reasons to eliminate them, which is reflected in our CHAM module in Figure 3.50. The Maude module
in Figure 3.51 defines the evaluation strategies of IMP’s constructs and should be now clear: in addition
to the syntactic details discussed above, it simply gives the Maude representation of the heating/cooling
rules in Figure 3.48. Finally, the Maude module in Figure 3.52 implements the semantic rules and the state
initialization heating rule in Figure 3.48, replacing the state airlock operation by multiset matching.

To test the heating/cooling rules, one can write Maude commands such as the two search commands
below, asking Maude to search for all heatings of a given syntactic molecule (the sort of X, Id, is already
declared in the last module):

    search { | X = 3 / (X + 2) ; | } =>! Sol:Solution .
    search { | X = (Y:Id + 2:Id) / (X + 2) ; | } =>! Sol:Solution .

The former gives only one solution, because heating can only take place on non-result subexpressions

Solution 1 (state 4)
states: 5 rewrites: 22 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> { | X > [] + 2 > 3 / [] > X = [] ; > [] | }
but the latter gives three solutions:

Solution 1 (state 5)
states: 8 rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {{ Y:Id \to [] + Z:Id \to [] / (X + 2) \to X = [] ; \to [] |}

Solution 2 (state 6)
states: 8 rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {{ Z:Id \to Y:Id + [] \to [] / (X + 2) \to X = [] ; \to [] |}

Solution 3 (state 7)
states: 8 rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {{ X \to [] + 2 \to (Y:Id + Z:Id) / [] \to X = [] ; \to [] |}

Each of the solutions represents a completely heated term (we used \(!\) in the search commands) and corresponds to a particular order of evaluation of the subexpression in question. The three solutions of the second search command above reflect all possible orders of evaluation allowed by our Maude semantics of CHAM(IMP). Interestingly, there is no order in which \(X\) is looked up in between \(Y\) and \(Z\). This is not a problem for IMP, but it may result in loss of behaviors for IMP++ programs. Indeed, it may be that other threads modify the values of \(X\), \(Y\), and \(Z\) while the expression above is evaluated by another thread in such a way that behaviors are lost if \(X\) is not allowed to be looked up between \(Y\) and \(Z\). Consequently, our orientation of the heating/cooling rules came at a price: we lost the fully non-deterministic order of evaluation of the arguments of strict operators; what we obtained is a non-deterministic choice evaluation strategy (an order of evaluation is non-deterministically chosen and cannot be changed during the evaluation of the expression—this is discussed in more depth in Section 3.3).

Maude can now act as an execution engine for CHAM(IMP). For example, the Maude command

\[
\text{rewrite } \{| \text{sumPgm } | \}.
\]

where \text{sumPgm} is the first program defined in the module IMP-PROGRAMS in Figure 3.4 produces a result of the form:

\[
\text{rewrite in TEST : } \{| \text{sumPgm } | \}.
\]

rewrites: 7543 in ... cpu (... real) (... rewrites/second)
result Solution: {{ | {| | |} #{| n |\to \emptyset & s |\to 5050 | |} |}}

Like in the previous Maude semantics, one can also search for all possible behaviors of a program using search commands such as

\[
\text{search } \{| \text{sumPgm } | \} \Rightarrow! \text{Sol:Solution}.
\]

Like before, only one behavior will be discovered (IMP is deterministic so far). However, an unexpectedly large number of states is generated, 4119 versus the 1709 states generated by the previous small-step semantics), mainly due to the multiple ways to apply the heating/cooling rules:

Solution 1 (state 4118)
states: 4119 rewrites: 12658 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {{ | {} |} #{| n |\to \emptyset & s |\to 5050 | |} |}

3.8.4 Notes

The chemical abstract machine, abbreviated CHAM, was introduced by Berry and Boudol in 1990 [8][9]. In spite of its operational feel, the CHAM should not be mistakenly taken for a variant of (small-step) SOS.
fact, Berry and Boudol presented the CHAM as an alternative to SOS, to address a number of limitations inherent to SOS, particularly its lack of true concurrency and what they called SOS’ “rigidity to syntax”. The basic metaphor giving its name to the CHAM was inspired by Banâtre and Le Métayer’s GAMMA language \[4, 5, 6\], which was the first to view a distributed state as a solution in which many molecules float, and the first to understand concurrent transitions as reactions that can occur simultaneously in many points of the solution. GAMMA was proposed as a highly-parallel programming language, together with a stepwise program derivation approach that allows to develop provably correct GAMMA programs. However, following the stepwise derivation approach to writing GAMMA programs is not as straightforward as writing CHAM rules. Moreover, CHAM’s nesting of solutions allows for structurally more elaborate encodings of data and in particular for more computational locality than GAMMA. Also, the CHAM appears to be more suitable as a framework for defining semantics of programming languages than the GAMMA language; the latter was mainly conceived as a programming language itself, suitable for executing parallel programs on parallel machines \[3\], rather than as a semantic framework.

The distinction between heating, cooling and reaction rules in CHAM is in general left to the user. There are no well-accepted criteria and/or principles stating when a rule should be in one category or another. For example, should a rule that cleans up a solution by removing residue molecules be a heating or a colling rule? We prefer to think of it as a cooling rule, because it falls under the broad category of rules which “structurally rearrange the solution after reactions take place” which we methodologically decided to call cooling rules. However, the authors of the CHAM prefer to consider such rules to be heating rules, with the intuition that “the residue molecules evaporate when heated” \[9\]. While the distinction between heating and cooling rules may have a flavor of subjectivity, the distinction between actual reaction rules and heating/cooling rules is more important because it gives the computational granularity of one’s CHAM. Indeed, it is common to abstract away the heating/cooling steps in a CHAM rewriting sequence as internal steps and then define various relations of interest on the remaining reaction steps possibly relating different CHAMS, such as behavioral equivalence, simulation and/or bisimulation relations \[8, 9\].

The technique we used in this section to reversibly and possibly non-deterministically sequentialize the program syntax by heating/cooling it into a list of computational tasks was borrowed from the K framework \[68, 66, 45, 64\] (also Section \[5.12\]). This mechanism is also reminiscent of Danvy and Nielsen’s refocusing technique \[19, 20\], used to execute reduction semantics with evaluation contexts by decomposing evaluation contexts into stacks and then only incrementally modifying these stacks during the reduction process. Our representation of the CHAM into rewrite logic was inspired from related representations by Serbanuta et al. \[74\] and by Meseguer \[12, 42\]. However, our representation differs in that it enforces the CHAM rewriting to only take place in solutions, this way being completely faithful to the intended meaning of the CHAM reactions, while the representations in \[74, 42\] are slightly more permissive, allowing rewrites to take place everywhere rules match; as explained, this is relevant only when the left-hand-side of the rule contains precisely one molecule.

We conclude this section with a note on the concurrency of CHAM rewriting. As already explained, we are not aware of any formal definition of a CHAM rewriting relation that captures the truly concurrent computation advocated by the CHAM. This is unfortunate, because its concurrency potential is one of the most appealing aspects of the CHAM. Moreover, as already discussed (after Theorem \[21\]), our rewrite logic representation of the CHAM is faithful only with respect to one non-concurrent step and interleaves steps taking place within the same solutions no matter whether they involve common molecules or not, so we cannot borrow rewrite logic’s concurrency to obtain a concurrent semantics for the CHAM. What can be done, 

---

12Meseguer \[42\] was published in the same volume of the Journal of Theoretical Computer Science as the CHAM extended paper \[9\] (the first CHAM paper \[8\] was published two years before, in 1990, same as the first papers on rewrite logic \[40, 39, 41\]).
however, is to attempt a different representation of the CHAM into rewrite logic based on ideas proposed by Meseguer in [43] to capture (a limited form of) graph rewriting by means of equational encodings. The encoding in [43] is theoretically important, but, unfortunately, yields rewrite logic theories which are not feasible in practice using the current implementation of Maude.

### 3.8.5 Exercises

**Exercise 168.** For any CHAM, any molecules $mol_1, \ldots, mol_k$, and any $1 \leq i \leq k$, the sequents

$$CHAM \vdash \{mol_1 \ldots mol_i\} \leftrightarrow \{mol_1 \triangleright \{mol_2 \ldots mol_i\} \triangleright mol_{i+1} \ldots mol_k\}$$

are derivable, where if $i = 1$ then the bag $mol_2 \ldots mol_i$ is the empty bag.

**Exercise 169.** Modify the CHAM semantics of IMP in Figure 3.45 to use a state data-structure as we did in the previous semantics instead of representing the state as a solution of binding molecules.

**Exercise 170.** Add a cleanup (cooling) rule to the CHAM semantics of IMP in Figure 3.45 to remove the useless syntactic subsolution when the computation is terminated. The resulting solution should only contain a state (i.e., it should have the form $\{\sigma\}$, and not $\{\{\sigma\}\}$ or $\{\cdot \{\sigma\}\}$).

**Exercise 171.** Modify the CHAM semantics of IMP in Figures 3.44 and 3.45 so that $\triangleright$ short-circuits when the numerator evaluates to 0.

**Hint:** One may need to make one of the heating/cooling rules for $\triangleright$ conditional.

**Exercise 172.** Modify the CHAM semantics of IMP in Figures 3.44 and 3.45 so that conjunction is non-deterministically strict in both its arguments.

**Exercise 173.** Same as Exercise 84 but for CHAM instead of big-step SOS: add variable increment to IMP, like in Section 3.8.2

★ Like for the Maude definition of big-step SOS and unlike for the Maude definitions of small-step SOS, MSOS and reduction semantics with evaluation contexts above, the resulting Maude definition can only exhibit three behaviors (instead of five) of the program $nondet++Pgm$ in Exercise 84. This limitation is due to our decision (in Section 3.8) to only heat on non-results and cool on results when implementing CHAMs into Maude. This way, the resulting Maude specifications are executable at the expense at losing some of the behaviors due to non-deterministic evaluation strategies.

**Exercise 174.** Same as Exercise 88 but for the CHAM instead of big-step SOS: add input/output to IMP, like in Section 3.8.2

**Exercise 175.** Same as Exercise 92 but for the CHAM instead of big-step SOS: add abrupt termination to IMP, like in Section 3.8.2

**Exercise 176.** Same as Exercise 104 but for the CHAM instead of small-step SOS: add dynamic threads to IMP, like in Section 3.8.2

**Exercise* 177.** This exercise asks to define IMP++ in CHAM, in various ways. Specifically, redo Exercises 114, 115, 116, 117, and 118 but for the CHAM of IMP++ discussed in Section 3.8.2 instead of its small-step SOS in Section 3.5.6

293
Add more citations here from others who used CHAMS to give semantics to different calculi or languages. Also, check whether the graph rewriting community has given the CHAM its desired concurrent semantics, using graph rewriting techniques. Ask Berry and Boudol. Also check with Corradini and Gadducci.

Add a new (sub)section on CHAM with env/store, like in the Maude code

Modify the CHAM definitions to dissolve statements, as we do in K

Add different types of molecules, and bags of them: for syntax, for state, for environments, location, etc.
3.9 Advanced Denotational Semantics: Continuations, Monads and Powerdomains

this should actually also discuss other advanced denotational semantics features, such as continuations and powerdomains
3.10 Discussion: Advantages, Disadvantages and Comparisons

don’t forget to say that big-step cannot even define a non-deterministic plus

have a discussion that syntactic purity may lead to annoying “encodings”, such as the one in Exercise 130.

discuss call/cc; we refer to it several times

One could argue that the changes above were necessary because of poor configuration management: instead of defining configurations as triples, one could have defined them as nested configurations of the form \( \langle s, \sigma, \omega \rangle \), and then only add the evaluation contexts “\( \text{Context} ::= \ldots \mid \langle \text{Context}, \text{List[Int]} \rangle \mid \text{output}(\text{Context}) \)” and the output rule \( \langle c, \omega \rangle \{ \text{output}(i) \} \rightarrow \langle c, (\omega, i) \} \} \) and change nothing else. Indeed, one would need to change nothing else because all the other rules would apply unchanged in contexts of the form \( \langle \square, \omega \rangle \).

3.11 Existing Approaches and Their Limitations for Programming Language Semantics

taken from JLAP’10 submission; needs to redistributed in sections above, as well as in Chapter on K

Since K was motivated by the need for a semantic framework satisfying the requirements discussed in Section 1, it is important to understand the limitations of the existing approaches as semantic frameworks for programming languages. In this section we group the relevant existing approaches into two categories: structural operational semantics frameworks and rewriting frameworks. For each approach we list its main characteristics, its limitations, and its relationship with K.

3.11.1 Structural Operational Semantics (SOS) Frameworks

We here discuss the most common SOS approaches, namely big-step and small-step SOS, their modular variant MSOS, and reduction semantics with evaluation contexts.

**Big-Step SOS**  Introduced as natural semantics in [34], also named relational semantics [49], or evaluation semantics, big-step semantics is “the most denotational” of the operational semantics. One can view big-step definitions as definitions of functions interpreting each language construct in an appropriate domain. Big-step operational semantics can be easily and efficiently interpreted in any recursive, functional or logical framework. It is particularly useful for defining type systems.

**Limitations:** Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred. Divergence is not observable in the evaluation relation. It may be hard or impossible to model concurrent features. It is not modular, e.g., to add side effects to expressions, one must redefine the rules to allow expressions to evaluate to pairs (value-store). It is inconvenient (and non-modular) to define complex control statements.

**Relationship to K:** None.
Small-step SOS  Introduced by Plotkin in [60,61], also called transition semantics or reduction semantics, small-step semantics captures the notion of one computational step. Therefore, it stops at errors, pointing them out and it is easy to trace and debug. It gives interleaving semantics for concurrency.

Limitations: Like big-step, it is non-modular. It does not give a “true concurrency” semantics, that is, one has to choose a certain interleaving (no two rules can be applied on the same term at the same time), mainly because reduction is forced to occur only at the top. It is still hard to deal with control—one has to add corner cases (additional rules) to each statement to propagate control changing information. Each small step traverses the entire program to find the next redex; since the size of the program can grow unbounded, each small step may take unbounded resources in the worst case.

Relationship to K: None.

Modular SOS  Mosses introduced modular SOS (MSOS) [51,52,53] to deal with the non-modularity of small-step and big-step SOS. The MSOS solution involves moving the non-syntactic state components as labels on transitions, plus a discipline of only selecting needed attributes from the states.

Limitations: Same as the ones above, except for modularity.

Relationship to K: Both MSOS and K make use of labeled information to achieve modularity. MSOS uses labels as record fields on the transition relation, while K uses labels as cell names in the configuration. In both MSOS and K one can use the labels in semantic rules to only refer to those items of interest.

Reduction semantics with evaluation contexts  Introduced by Felleisen and colleagues (see, e.g., [25,88]), the evaluation contexts style improves over small-step SOS in two ways: (1) it gives a more compact semantics to context-sensitive reduction, by using parsing to find the next redex rather than small-step rules; and (2) it provides the possibility of also modifying the context in which a reduction occurs, making it much easier to deal with control-intensive features. Additionally, one can also incorporate the configuration as part of the evaluation context, and thus have full access to semantic information “by need”.

Limitations: It still only allows “interleaving semantics” for concurrency. It is too “rigid to syntax”, in that it is hard or impossible to define semantics in which values are more than plain syntax; for example, one cannot give a syntactic semantics to a functional language based on closures for functions (instead of substitution), because one needs special, non-syntactic means to handle and recover environments (as we do in K, see Sections FIXTHIS ). Although context-sensitive rewriting might seem to be easily implementable by rewriting, in fact one has to perform an amount of “parsing” work linear in the size of the program for each computational step, like in small-step SOS. However, one might obtain efficient implementations for restricted forms of context-reduction definitions by applying refocusing techniques [19].

Relationship to K: Both reduction semantics and K make use of evaluation contexts and can store or modify them. Reduction semantics relies on a mechanism to “split/plug” syntax into contexts, while K relies on a mechanism to “heat/cool” syntax into computations. The former is achieved by an implicit “advanced” parsing of syntax into a context and a redex, while the latter is achieved using rewrite rules.

3.11.2 Rewriting Frameworks

We here discuss some rewriting approaches which have been used or have the potential to be used for programming language semantics.

Term Rewriting and Rewrite Logic  Rewrite logic was proposed by Meseguer [42] as a logical formalism generalizing both equational logic and term rewriting, so it more suitable for language semantics than plain
term rewriting. The rewrite logic semantics (RLS) project [46, 48] is an initiative aiming at using rewrite logic for various semantic aspects of programming languages; a substantial body of work on RLS, both of theoretical and practical nature, is discussed in [46, 48], which we do not repeat here. It is worth mentioning that all the various SOS semantic approaches in Section 3.11.1 (and others not discussed here) can be framed as particular definitional styles in rewriting logic and then executed using Maude [74].

Limitations: Rewrite logic is a general-purpose computational framework, which provides the infrastructure but no hints on how to define a language semantics as a rewrite system. Definitional styles need to be represented within rewriting as shown in [74], such as those in Section 3.11.1. Additionally, rule instances cannot overlap, which results in an enforcement of interleaving in some situations where one would like to have true concurrency. For example, two threads reading from a shared store have to interleave the store operations even if they act on different locations, because the corresponding rule instances overlap on the store; some hints are given in [43] on how to use equations in an ingenious way to still achieve theoretical true concurrency in the presence of sharing, but that approach is impractical for language semantics.

Relationship to K: Like rewrite logic, K is a computational framework that aims at maximizing the amount of concurrency that can be achieved by means of term rewriting. However, one K concurrent rewrite step may take several interleaved rewrite logic steps.

Graph Rewriting Graph rewriting [22] extends the intuitions of term rewriting to arbitrary graphs, by developing a match-and-apply mechanism which can work directly on graphs.

Limitations: Graph rewriting is rather complex and we are not aware of works using it for programming language semantics. Part of the reason could be the notorious difficulty of graph rewriting to deal with structure copying and equality testing, operations which are crucial for programming language semantics and relatively easy to provide by term rewriting systems. Finally, in spite of decades of research, at our knowledge there are still no graph rewriting engines comparable in performance to term rewrite engines.

Relationship to K: The K rules, like those in graph-rewriting, have both read-only components, which could be shared among rule instances to maximize concurrency, and write-only components.

The Chemical Abstract Machine Berry and Boudol’s chemical abstract machine [9], or CHAM, is both a rewrite-based model of concurrency and a specific style of giving language semantics. The CHAM views a distributed state as a “solution” in which many “molecules” float, and understands concurrent transitions as “reactions” that can occur simultaneously in many points of the solution.

Limitations: While chemistry as a model of computation sounds attractive, technically speaking the CHAM is in fact a restricted case of rewrite logic, as noticed in [42]. Moreover, some of the chemical “intuitions”, such as the airlock operation which imposes a particular chemical “discipline” to access molecules inside a solution, inhibit the potential for the now well-understood and efficient matching and rewriting modulo associativity and commutativity. In fact, at our knowledge, there is no competitive implementation of the CHAM. Although this solution-molecule paradigm seems to work for languages with simple state structure, it is not clear how one could represent the state for complex languages with threads, locks, environments, etc. Finally, CHAMs provide no mechanism to freeze the current molecular structure as a “value”, and then to store or retrieve it, as we would need to define language features like call/cc. It would therefore seem hard to define complex control-intensive language features in CHAM.

Relationship to K: Like the CHAM, K also organizes the configuration of the program or system as a potentially nested structure of molecules (called cells and potentially labeled in K). Like in CHAM, the configuration is also rewritten until it “stabilizes”. Unlike in CHAM, the K rules can match and write inside and across multiple cells in one parallel step. Also, unlike in CHAM (and in rewrite logic), K rewrite rules
can apply concurrently even in cases when they overlap (read-only structure).

**P-Systems**  Päun’s membrane systems (or P-systems) \[^{58}\] are computing devices abstracted from the structure and the functioning of the living cell. In classical *transition P-systems*, the main ingredients of such a system are the *membrane structure*, in the compartments of which *multisets* of symbol-objects evolve according to given *evolution rules*. The rules are localized, associated with the membranes (hence, with the compartments), and they are used in a *nondeterministic maximally parallel* manner.

**Limitations:** When looked at from a programming language semantics perspective, the P-systems have a series of limitations that have been addressed in K, such as: (1) Lack of structure for non-membrane terms, which are plain constants; and (2) Strict membrane locality (even stricter than in the CHAM), which prevents rules to match and rewrite within multiple membranes at the same time. Regarding (1), programming languages often handle complex data-structures or make use of complex semantic structures, such as environments mapping variables to values, or closures holding code as well as environments, etc., which would be hard or impossible to properly encode using just constants. Regarding (2), strict membrane locality would require one to write many low-level rules and introduce and implement by local rules artificial “cross-membrane communication protocols”. For example, the semantics of variable lookup involves acquiring the location of the variable from the environment cell (which holds a map), then the value at that location in the store cell (which holds another map), and finally the rewrite of the variable in the code cell into that value. All these require the introduction of encoding constants and rules in several membranes and many computation steps; compare that to the one step K rule in Section FIXTHIS.

**Relationship to K:** K-systems share with P-systems the ideas of cell structures and the aim to maximize concurrency. However, K rules can span across multiple cells at a time, and the objects contained in a cell can have a rich algebraic structure (as opposed to being constants, as in P-systems). \[^{75}\] presents an in-depth comparison between the two formalisms, showing how one can use K to model P-systems.

### 3.11.3 old limitations stuff about big-step and small-step SOS

**Big-Step vs. Small-Step: Advantages/Disadvantages**

The theorem above tells us that the two styles of semantics achieve eventually the same objective, at least for the simple language that we considered: they tell formally when a program \( p \) evaluates to an integer \( i \). So when do we choose one versus the other when we define a programming language? To answer this question, one needs to reflect on and to understand the advantages and the disadvantages of each of the two definitional styles.

**Advantages of Big-Step SOS.**

- When it can be given to a language, it is *easy to understand* since it relates syntactic entities directly to their expected results.
- Because of its recursive nature, big-step SOS gives a strong intuition on how to *implement an interpreter* for a language. Many proponents of big-step SOS regard it “almost as an interpreter” for the defined language.
- *More abstract*, more mathematical/denotational; therefore, one can more easily define and *prove properties* about programs; of course, because of the equivalence of the two semantics, one can also use the transitive closure of the small-step SOS transition relation, but this may be less natural than using the big-step semantics. For example, try to prove the following using both big-step and small-step semantics:

**Exercise 178.** Formulate using big-step and small-step SOS, respectively, what it means for the statements while \( b \) (while \( b \) \( s \)) and while \( b \) \( s \) to be “equivalent”, and then prove it.
- Particularly useful when defining type systems of programming languages; there, values are replaced by their types.

**Disadvantages of Big-Step SOS.**
- As given, big-step SOS is not executable: one always needs to provide a result value or state to each language construct and then use the SOS rules to “check” whether that result value or state is indeed correct. Even though in practice one can often implement an interpreter relatively easily by disseminating the big-step SOS rules of a language, they are in fact intended to give a formal description of what is possible in a language, not to be executable “per se”.
- Because of the above, big-step SOS avoids or hides non-termination of programs by avoiding or hiding entirely the means by which one can state that a program does not terminate. Non-termination appears as “lack of proof” in the SOS system, but so does a program that performs a division by zero (or, in more complex languages, one that produces any runtime error); it is fair to say that, in principle, “runtime errors” can be handled in big-step SOS by generating and propagating “error values”.
- Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred.
- Most importantly, it is widely accepted that it is very inconvenient or impossible to define nontrivial concurrent languages using big-step SOS. It is fair to say that big-step SOS provides some limited support for handling non-determinism. For example, one can add a Boolean constant $\texttt{randBool}$ that non-deterministically evaluates to either $\texttt{true}$ or $\texttt{false}$ as follows:

\[
\begin{align*}
\langle \texttt{randBool}, \sigma \rangle \Downarrow & \langle \texttt{true} \rangle \\
\langle \texttt{randBool}, \sigma \rangle \Downarrow & \langle \texttt{false} \rangle
\end{align*}
\]  

The big-step SOS definition of our language extended with the rules above can now relate programs with any of the integer values that they non-deterministically evaluate to.

Unfortunately, the situation is more complex when the non-determinism is a consequence of parallelism. Consider, for example, the addition of a language construct $s_1 || s_2$ for evaluating statements $s_1$ and $s_2$ in parallel. If one naively adds rules like

\[
\begin{align*}
\langle s_1, \sigma \rangle \Downarrow & \langle \sigma'' \rangle, \langle s_2, \sigma'' \rangle \Downarrow \langle \sigma' \rangle \\
\langle s_1 || s_2, \sigma \rangle \Downarrow & \langle \sigma' \rangle \\
\langle s_2, \sigma \rangle \Downarrow & \langle \sigma'' \rangle, \langle s_1, \sigma'' \rangle \Downarrow \langle \sigma' \rangle \\
\langle s_1 || s_2, \sigma \rangle \Downarrow & \langle \sigma' \rangle
\end{align*}
\]  

then one fails to capture the intended non-determinism in our parallel language because one simply sequentializes the executions of the two statements instead of executing them in parallel.

**Exercise 179.** Give an example of a parallel program using the parallel statement construct above for which a big-step SOS definition like above would miss some possible results.

To correctly and completely capture the desired non-determinism of this parallel language, one needs to monolithically detect the first actions that the parallel program can potentially take, then, for a given such action, to perform it collecting the new global state after its execution, and then to generate a big-step configuration comprising a new parallel program. Complications are mostly due to the fact that a big-step semantics for this concurrent language, in case any can be given by some hard dying proponent of big-step SOS, would not be compositional. For example, there is no way to fully characterize the derivability of
\[ \langle s_1; s'_1\rangle \parallel \langle s_2, \sigma \rangle \Downarrow \langle \sigma' \rangle \] in terms of the derivability of any group of transitions involving only the statements \( s_1 \), \( s'_1 \) and \( s_2 \), such as \( \langle s_1; s'_2, \sigma \rangle \Downarrow \langle \sigma'' \rangle \), etc.

**Advantages of Small-Step SOS.**
- It is executable. Indeed, one can start with a program to evaluate and keep applying SOS rules whose left configuration matches; to apply a small-step rule, one typically needs to call recursively the “execution” procedure on smaller fragments (the ones above the lines). Nevertheless, one can log all the successful applications of rules obtained this way and thus get an execution of the original program. Therefore, it is easy to trace and debug. It is relatively straightforward to implement an interpreter for a language by just following, almost blindly, a small-step SOS definition of that language.

- Thanks to the above, non-termination of programs results in non-termination of searching for a proof; however, programs that perform division by zero (or runtime errors in general) can still be evaluated step-by-step until the actual error takes place; then one can be given a meaningful error message, including the entire stuck program.

- It supports definitions of non-deterministic and/or parallel languages, obeying the interleaving semantics approach to concurrency. In fact, our small-step SOS definition of the language above was non-deterministic; it just “happened” that the final results of evaluating expressions or programs were deterministic (but one needs to prove it). Considering the parallel statement construct \( s_1; s_2 \) discussed in the context of big-step SOS above, one can easily give it a small-step SOS semantics as follows:

\[
\begin{align*}
\langle s_1, \sigma \rangle & \rightarrow \langle s'_1, \sigma' \rangle \\
\langle s_1; s_2, \sigma \rangle & \rightarrow \langle s'_1; s_2, \sigma' \rangle \\
\langle s_2, \sigma \rangle & \rightarrow \langle s'_2, \sigma' \rangle \\
\langle s_1; s_2, \sigma \rangle & \rightarrow \langle s_1; s'_2, \sigma' \rangle
\end{align*}
\] (3.10)

(3.11)

**Disadvantages of Small-Step SOS.**
- It is somehow too low level and explicit, which results into relatively large small-step language definitions; in particular, one needs to explicitly give all the “congruence” rules for ordinary operators such as addition, multiplication, etc.

- Small-step SOS gives only an interleaving semantics to a concurrent language, that is, it regards executions or behaviors of concurrent programs as linear lists of actions.

Small-step SOS is inherently limited with regards to giving concurrent systems a true concurrency semantics, because it enforces reduction to occur only at the top, thus disallowing concurrent applications of rules. However, on parallel architectures, executions of concurrent programs are not interleaved; for example, statements that have only local effect in different processes can be and typically are safely executed concurrently.

- Less suitable for proving properties about programs; however, if one can also give a big-step semantics of the same language and prove it equivalent to the small-step semantics, then one can perform proofs using the big-step one.

**Disadvantages of both Big-Step and Small-Step SOS**
- They are not modular, in the sense that the addition of new statements to a language may require one to change many other rules corresponding to definitions of unrelated statements.

**Exercise 180. (20 points)** Let us extend our sequential language with two common features, namely variable increment, which increments the variable (right away) and returns its new value in context, and halt, which
takes an arithmetic expression, evaluates it to an integer and then stops the program with that value returned as final result:

\[
\text{AExp} ::= \ldots | \text{++Var} \\
\text{Stmt} ::= \ldots | \text{halt AExp}
\]

1. **(4 points)** Add variable increment to the big-step SOS: list all the new rules, as well as the new versions of all the already existing rules that need to change;

2. **(2 points)** Do the same for small-step SOS;

3. **(4 points)** Add halt to the original big-step SOS definition (forget the increment for the time being): list all the new rules, as well as the new versions of all the already existing rules that need to change;

4. **(4 points)** Do the same for small-step SOS;

5. **(4 points)** Add both variable increment and halt to the original big-step and small-step SOS definitions: for each of them list all the new rules, as well as the new versions of all the already existing rules that need to change;

For small-step SOS, pick one of the two variants discussed in the lecture notes.

**Comment on why we had to make the various changes and say which of big-step and small-step SOS semantics appears to you to be more modular and why.**

(2) **Big-step and small-step SOS do not properly capture the intended computational granularity as deduction steps.** The original motivation of SOS semantics, both big-step and small-step, was to offer a mathematical, logical framework in which the otherwise purely intuitive notion of “computation” in a programming languages becomes logical deduction, or inference, in the formal SOS definition of the corresponding programming language. While one can argue that SOS proofs properly capture the final result of program computations, they do not properly capture the intended computational granularity of the defined language features.

For example, the intended computational granularity of a “halt a” statement is “evaluate a and then stop the execution of the program with that value”; compare that with “evaluate a, then propagate its value all the way back to the top over the syntax of the executing program, and then stop the program with that value”, which is what happens in both big-step and small-step SOS.

A less important violation of computation granularity appears also in the small-step SOS rule for loops: a while loop transitions in one-step into a conditional statement. That wasted step was not originally intended; it is just an artifact of the fact that everything in SOS needs to be a rule. What one would like to say is that that unrolling of while loops is nothing but a structural identity, or an unobservable transition, with no intended computational content.

(3) **Somehow related to the previous two disadvantages, it is in general hard or impossible to define control-intensive language features in SOS.** For example, if one adds functions to our language together with a return statement, then one either needs to traverse back the syntax when a value is returned until the corresponding function call is encountered, thus violating again the desired computational granularity of return, or otherwise needs to add a function stack to the configurations. In the latter case, not only that all the configurations will change having to include the stack, so all the existing rules will have to change thus breaking the modularity of the definition, but more importantly, one needs to grab the current “evaluation context” when a function is invoked in order to push it in the stack, so one can recover it when the function returns. Unfortunately, “evaluation context” are captured by “proof context” in standard SOS (we will
shortly see that Felleisen’s reduction semantics with evaluation contexts will make contexts explicit), and proof contexts are not first class objects in SOS derivations that can be stored and retrieved at will. The list of “SOS-inconvenient” features that can be added to the language can of course continue: exceptions, break/continue in loops, etc.

(4) *Neither big-step nor small-step SOS provides an appropriate semantical foundation for concurrent languages.* Big-step SOS can hardly be used to define any meaningful concurrent language, while small-step SOS gives only an interleaving semantics.

(5) They are both operational and syntax-driven, so they tell us close to *nothing about models* of languages. Models, which in SOS are considered “something else”, are related to both denotational semantics and realizations of languages, including implementations for them.

3.11.4  **old stuff about MSOS**

**Advantages of MSOS**

1) *MSOS adds modularity to SOS definitions* by allowing one to refer only to configuration items of interest in each rule and using “...” for the remaining ones. This way, adding rules for new language features that need additional configuration infrastructure can be done without having to modify any of the rules for unrelated existing features.

**Exercise 181.** (10 points) *Add rules for halt a (this new statement was explained in the previous homework exercise) to the MSOS definition above. How many rules did you need? How many existing rules did you have to change?*

(Hint: you may need to add one more field in labels to store the “halting signal”; feel free to also change the syntax of the language however you wish, in particular to add one more top-level program construct if you need to “catch” the halting signal).

2) By allowing unobservable transitions, MSOS provide better support for allowing the language designer to tune the computation granularity of defined language features.

3.11.5  **old stuff about context reduction**

**Advantages of Context Reduction**

1) *Evaluation contexts are explicit first-class entities* in context reduction, so they can be modified, passed, stored and retrieved. This allows for elegant definitions of control intensive language features such as abrupt termination of programs using halt-like constructs (just dissolve the current evaluation context), functions with return (stack the evaluation contexts at function calls and restore them at return), exceptions, and even callcc (store the current evaluation context as a value in the store, retrieve it when applied). Moreover, these operations on contexts can be done in such a way that they have the expected computation granularity.

**Exercise 182.** (10 points) *Add context reduction definitions for increment and halt (same language constructs as in the first homework exercise). Give definitions using both context reduction styles described above.*

2) Since reductions can only happen in proper evaluation contexts and propagation of computation is done once and for all using the context reduction rule, one needs not worry about propagating local computations up-wards along the syntax; this way, definitions in context reduction become *significantly more compact* than SOS definitions.
Disadvantages of Context Reduction

1) Context reduction is based on an implicit mechanism to split a program or a fragment of program \( p \) into an evaluation context \( c \) and a fragment of program, also called redex, \( e \), so that \( p = c[e] \). This split operation is somehow assumed atomic and non-computational, so it elegantly captures and hides all the small-step propagation rules of SOS. However, this decomposition is not obvious and may require significant computational resources, sometimes linear in the size of the program. Moreover, since a new evaluation context needs to be recalculated at almost any reduction step, it can easily become the major bottleneck in the efficiency of an interpreter implemented following the context reduction approach. While such an interpreter would be no worse than one implemented following the small-step SOS approach, it can still be a lot slower than it needs to be. Techniques such as refocusing have been proposed to incrementally compute the next evaluation context, but these techniques appear to work only when the decomposition of the program into an evaluation context and a redex is deterministic.

2) It is somehow still awkward to define concurrent languages using context reduction. That is because concurrent processes typically communicate with each other from different places in the configuration, so one may need contexts with more than one hole, preferably with an arbitrary number of them. Computing and matching such such contexts on complex configurations becomes a significantly non-trivial issue.

3) It still does not serve as a solid foundation for concurrency, because, just like SOS, it only captures an interleaving semantics of a defined concurrent system.

4) It still says nothing about models, being essentially a purely syntactic reduction technique.

We are going to shortly discuss a language definitional technique, called K, which has none of the limitations above. Moreover, we’ll present an automatic translation from any context reduction definition, say CR, into a K definition, say k(CR), which is faithful to the computational granularity of the original context reduction definition, in the sense that any computational step in CR has a precise one-step correspondent in k(CR) and vice-versa, that is, any computational step in k(CR) has a precise correspondent step in CR. Additionally, k(CR) is as compact as the original CR: it adds one equation per context evaluation production and one K-rule per context reduction rule. Therefore, if one wants to use context reduction within K then one is free to do so, but, of course, one cannot circumvent the limitations of context reduction. As shown in the later classes by a series of languages defined in K, such as imperative, functional, object-oriented and logic programming, as well as concurrent languages, one may be better off using the full strength of K, unlimited by restricted methodological fragments of K, such as context reduction.

Disadvantages of MSOS

While modularity was a major drawback of conventional SOS and MSOS proposes solutions to improve it, MSOS still carries most of the other disadvantages of SOS. For example, it still has an interleaving semantics for concurrency and it still says nothing about models. While providing better support for defining the desired computational granularity of language constructs by allowing certain transitions to be unobservable, MSOS still needs to propagate information through the program syntax in order for that information to reach its destination. For example, in the case of halt, the “halting signal” still had to be propagated from subexpressions to parent expressions in labels; it is true that the program did not need to be changed anymore like in the case of small-step SOS, but the number of rule applications remains pretty much the same as in small-step SOS; therefore, we still cannot model the intended computational granularity of halt \( a \), which is “evaluate \( a \) to \( i \) and then stop immediately the execution of the entire program with result \( i \). We will shortly see how reduction semantics with evaluation contexts allows us to do that.

Like conventional SOS, MSOS is still rather low-level, mimicking by logical inference both one-step computations and propagation of reductions along the language constructs. That means that computation contexts are captured as proof-contexts and, unfortunately, proof contexts cannot be stored and/or retrieved at
will. That means that complex control-intensive language constructs like \texttt{callcc} may be impossible to define in MSOS.

A disadvantage of MSOS compared to SOS is that it is even more syntactic in its nature than SOS: every transition must contain pieces of syntax in both sides. Recall our first small-step SOS variant, which allowed us to derive transitions of the form \( \langle s, \sigma \rangle \to \langle \sigma' \rangle \) when the statement \( s \) carried no further computation. To write that in MSOS, one would need to first generate an artificial “statement value” and then discard it with another rule. One could find that natural to do, especially in the context of a functional programming language, but on the other hand one can argue that one should not be enforced to produce an artificial value for things that are commonly used exclusively for their side effects.

### 3.11.6 other old stuff about context reduction

The rule for \texttt{halt} in Version 1 could have also been \( "c[\texttt{halt } i] \to i" \), though it would yield some perhaps undesired top-level nondeterminism: a configuration parsed as \( \langle c[\texttt{halt } i], \sigma \rangle \) could indeed reduce to \( \langle i, \sigma \rangle \) by applying the characteristic rule of context reduction in the context \( \langle \square, \sigma \rangle \), but it could also reduce directly to \( i \) because an alternative parsing of it is \( "c, \sigma)[\texttt{halt } i]" \), thus allowing top-level reductions between terms of different syntactic categories.

**Strengths.** Context reduction improves over SOS in at least two ways:

1. Since reductions can only happen in proper evaluation contexts and propagation of computation is done once and for all using the characteristic rule of context reduction, one needs not worry about propagating local computations up-wards along the syntax; this way, definitions in context reduction become more compact than SOS definitions. In simple terms, context reduction yields more compact definitions than ordinary SOS because it uses parsing to find the next redex rather than towered small-step rules.

2. It provides the possibility of also altering the evaluation context in which a reduction occurs. More precisely, unlike SOS, context reduction allows the possibility to handle evaluation contexts like any other entities in the language, in particular to pass them to other contexts or to store them. This way one can easily and elegantly define the semantics of control-intensive language features such as return of functions, break/continue of loops, exceptions, and even \texttt{callcc}.

As shown shortly, context reduction can be faithfully translated (i.e., nothing is lost in translation) into rewrite logic and even in a small fragment of K, replacing each context reduction rule by a corresponding rewrite logic or K rule. That implies that language definitions in K are provably at least as powerful and compact as those using context reduction. However, when using the unleashed computation handling capabilities of K by disobeying what was called in \[9\] “the rigidity of syntax”, one can often write even more compact, and in our (admittedly subjective) view more natural, language definitions.

**Weaknesses.** The advantages of context reduction over SOS-like approaches, namely the compactness of definitions and the explicit control over evaluation contexts, come at a price. We next discuss the drawbacks of context reduction, first by comparison with SOS and then in general.

The discussion above may imply that context reduction is strictly superior to SOS or other syntactic approaches. That is, unfortunately, not true. SOS’ conditional rules have an important advantage, which cannot be recovered in context reduction: they easily and naturally create execution environments just for the subexpressions (or other fragments of program) to be processed as part of the condition, without having to
worry about recovering the execution environment when the processing of the subexpression is finished. For example, recall the MSOS definition (one can obtain an equivalent SOS definition by desugaring MSOS into SOS \cite{63}) of let in Figures ?? and ??,

\[
\begin{align*}
    e_1 & \rightarrow e'_1 \\
    \text{let } x = e_1 \text{ in } e_2 & \rightarrow \text{let } x = e'_1 \text{ in } e_2 \\
    e_2 & \xrightarrow{\text{env}=\rho[v_1/x]...} e'_2 \\
    \text{let } x = v_1 \text{ in } e_2 & \xrightarrow{\text{env}=\rho...} \text{let } x = v_1 \text{ in } e'_2 \\
    \text{let } x = v_1 \text{ in } v_2 & \rightarrow v_2
\end{align*}
\]

These rules say that the binding expression \( e_1 \) must be first evaluated to a value \( v_1 \), which is then bound to \( x \) in the environment while the body of the let, \( e_2 \), is being reduced; the value \( v_2 \) that \( e_2 \) hereby eventually reduces to is the result of the evaluation of the let. The important thing to notice here is the second rule: \( e_2 \) is reduced in one step to \( e'_2 \) in a modified environment \( (\rho[v_1/x]) \), while \( \text{let } x = v_1 \text{ in } e_2 \) reduces to \( \text{let } x = v_1 \text{ in } e'_2 \) under the original environment \( (\rho) \). When using SOS or MSOS, the language designer is therefore not concerned with “recovering” the environment after the reduction of \( e_2 \). A similar scenario happens when giving an SOS semantics of function invocation in an environment-based definition. Unfortunately, neither of these is possible in context reduction. Even though the third rule above could be used as is in a hypothetical context reduction definition and the first rule can be replaced by an evaluation context production giving evaluation access to the binding expression,

\[ Cxt ::= \ldots | \text{let } Id = Cxt \text{ in } E, \]

there is no way to find an equivalent for the second rule in context reduction. What one would like to say is, using an unconditional rule, that in an expression of the form \( \text{let } x = v_1 \text{ in } e_2 \) evaluation access is given to \( e_2 \), but in the modified environment \( (\rho[v_1/x]) \); then, once \( e_2 \) is completely reduced, the environment changes back to \( \rho \). This is simply not possible in context reduction, unless one modifies the syntax in some rather aggressive way (introducing, e.g., explicit environment restore statements), which would lack generality and be against the “syntactic” spirit of context reduction.

Also, one should not be fooled thinking that context reduction is a generalization of SOS, so one can use conditional SOS-like rules as an escape whenever a particular task cannot be accomplished using unconditional context reduction rules. For example, one cannot use a rule like the second one above, simply because one “loses the context” in the condition; indeed, if \( e_2 \) reduces to a call/cc during the processing of the condition, then one cannot provide it with the whole evaluation context. How does K solve these problems, considering that it is also an unconditional-rule framework? The answer is that K, by its nature, explicitly disobeys the purity of syntax; in the case of let, one uses the computation structure to perform the desired tasks in the desired order (that is the precise role of the computation structure): bind \( x \) to \( v_1 \), schedule \( e_2 \) for processing, then recover the original environment. Similar environment recovering steps need to be taken when defining control-changing statements such as return of functions, exceptions, or call/cc. Neither of these can be defined in context reduction following an environment-based definitional style.

The designated approach in context reduction to avoid the complications discussed in the paragraph above is to discard the environment-based style all together and follow instead a substitution-based style. Substitution is the basis of \( \lambda \)-calculus’ \( \beta \)-reduction rule and can be used to define other common language constructs, such as let and letrec. However, some language constructs that have straightforward environment-based definitions, such as references, objects, threads, etc., require quite intricate and tricky substitution-based
definitions, if any can be found. While substitution-based semantic techniques are interesting in their own way and are worth considering in some practical cases, we believe that a language designer should simply not be forced by the underlying framework to follow a substitution-based style. Moreover, an executable definitional framework should not disregard efficiency, and substitution is expensive. An executable language definition in which each computational step has a complexity linear in the size of the program is simply impractical (wrt executability); not to mention that the size of the program to reduce can grow unbounded in a substitution-based framework. As seen in Section ??, K also allows substitution-based definitions which are as simple and natural, if not simpler and more natural, than their corresponding context reduction definitions. However, K does not enforce, encourage or discourage substitution-based definitions. We believe that that should be the choice of the language designer and not imposed by the framework.

We next discuss several other weaknesses of context reduction, not necessarily by comparison with SOS.

- Context reduction is based on an implicit mechanism to split a program or a fragment of program \( p \) into an evaluation context \( c \) and a fragment of program, also called redex, \( e \), so that \( p = c[e] \). This split operation is somehow assumed atomic and non-computational, so it elegantly captures and hides all the small-step propagation rules of SOS. However, this decomposition is not obvious and may require significant computational resources, sometimes linear in the size of the program or worse. Moreover, since a new evaluation context needs to be recalculated at almost any reduction step, it can easily become the major bottleneck in the efficiency of an interpreter implemented following the context reduction approach. While such an interpreter would be no worse than one implemented following the small-step SOS approach, it can still be a lot slower than it needs to be, thus making context reduction unattractive as an executable language definitional framework (see the “ideal framework” requirements in Section 1). Techniques such as refocusing [19] have been proposed to incrementally compute the next evaluation context, but these techniques appear to work only when the decomposition of the program into an evaluation context and a redex is deterministic.

- It is inconvenient to define concurrent languages using context reduction. That is because concurrent processes typically communicate with each other from different places in the configuration, so one may need contexts with more than one hole, preferably with an arbitrary number of them. While some attempts to define and handle multi-contexts have been proposed (see, e.g., [FIXTHIS]), computing and matching such contexts on complex configurations becomes a significantly non-trivial issue with potentially serious impact on performance.

- Context reduction is still far from serving as a solid foundation for concurrency, because, just like SOS, it only captures an interleaving semantics of a defined concurrent system. In particular, context reduction can only be used to give an interleaving semantics of CCS (middle column in Figure ??), but not a truly concurrent one (right column in Figure ??).

- It still says nothing about models, being essentially a purely syntactic reduction technique.

K has all the strengths of context reduction, at the same time avoiding its limitations. Indeed, K’s matching is comparatively cheap and rather standard in term rewriting; existing advanced indexing techniques
in term rewriting can be and are being used (by running K in Maude, for example) to efficiently execute K definitions. K is particularly useful to define concurrent languages, because, if used properly, can give a truly concurrent semantics to the defined languages at the same time also allowing an interleaving semantics—K’s approach is to let the language designer, rather than the limitations of the framework, decide what semantics for concurrency is desired. Also, K has a model-theoretical semantics, for the time being borrowed via its desugaring into rewrite logic.

3.11.7 older stuff about cham

When Cham definitions follow the style in Table FIXTHIS, i.e., taking a context-reduction-like approach, one could use as evaluation strategies heating only on redexes and cooling only on values, which would lead to a deterministic abstract-machine. Moreover, one can notice that airlock rules were introduced to select elements from a set without specifying the rest of the set, abstracted by a molecule. Efficient implementations should probably do exactly the opposite, that is, matching in the sets. To do that in our rewrite framework, one would orient the airlock rules in the sense of inserting back the “airlocked” molecules into their original solution and to apply them on the terms of the existing rules, to make the definition executable. The only
rules changing in the definition above are those involving the store; for example, the assignment rule will be transformed into:

$$\{ X := I \uplus C \}, \{(X, I'), \sigma\} \rightarrow \{ \{ \} \uplus C \}, \{(X, I), \sigma\}$$

One should notice that the specification obtained by these transformations is equivalent to the initial one, since it does not change the equivalence classes and the transitions. The main advantage of the newly obtained specification is that it is also executable in a deterministic fashion, that is, there is no need to search for a solution anymore.

**Strengths.** Being a special case of rewrite logic, it inherits many of benefits of rewrite logic, being specially well-suited for describing truly concurrent computations and concurrent calculi.

**Weaknesses.** Heating/cooling rules are hard to implement efficiently in general—an implementation allowing them to be bidirectional in an uncontrolled manner would have to search for solutions, possibly leading to a combinatorial explosion. Rewriting strategies such as those in [11, 85] can be of help for solving particular instances of this problem. Although this solution-molecule paradigm seems to work pretty well for languages in which the structure of the state is simple enough, it is not clear how one could represent the state for complex languages, with threads, locks, environments, and so on. Finally, Chams provide no mechanism to freeze the current molecular structure as a "value", and then to store or retrieve it, as we would need in order to define language features like \texttt{callcc}. Even though it was easy to define \texttt{halt} because we simply discarded the entire solution, it is hard or impossible to define more complex control-intensive language features in Cham.

3.12 K
3.13 Notes

This language is reminiscent of the Imp language defined by Winskel in his book

Chapter 4

Idealized Programming Languages, Features, and Models of Computation

In this part of the course we discuss two important and closely related mathematical theories:

- **Lambda calculus**, written also *λ*-calculus, is a pure calculus of functional abstraction and function application, with many applications in logic and computer science;

- **Combinatory logic** shows that bound variables can be entirely eliminated without loss of expressiveness. It has applications both in the foundations of mathematics and in the implementation of functional programming languages.

A good reference for these subjects is the book “The Lambda Calculus: Its Syntax and Semantics” by H.P. Barendregt (Second Edition, North Holland 1984). This book also contains a great discussion on the history and motivations of these theories.
4.1 Combinatory Logic

Combinatory logic was proposed by Mosses Schönfinkel in 1920, as a means to reduce the number of primitive notions of logic, in particular to eliminate the need of variables in functions and in predicates. Looking at it through programming language lenses, combinatory logic is a very simple language in which programs can be described as a combination of a small number of primitive elements, called combinators, and of free variables. Even though combinatory logic defines no explicit functions, each combinator can be thought of as a closed function, that is, a function which declares all the variables that it uses, and their combination can be thought of as function application\(^1\).

Combinatory logic can capture the meaning of any arithmetic or logical statement (and by extension, any non-interactive computer program), making it a Turing-complete computational model. It is often described and/or implemented in terms of rewriting (of graphs, trees, or terms).

Combinatory logic was proposed by Mosses Schönfinkel in 1920.

- who proposed it and why
- what is the idea
- formally what it is: the equational theory
- define other combinators: X, I, Ω, Y, Composition. Prove SKK=I under eta, etc. SII = Ω. Prove that if F = S (K f) (S I I) then F F is a fixed point for f. Then

result Exp: S (S (S (K S) (S (K K) (S K K))) (S (S (K S) (S (S (K S) (K K)) (K K))) (S (S (K S) (S (K S) (K K)) (K K)))) (S (S (K S) (S (K S) (K K)) (K K))) (S (S (K S) (S (K S) (K K)) (K K))))

is a fixed point combinator.

More precisely, we show that any closed \(\lambda\)-expression can be systematically transformed into a \(\lambda\)-expression build over only the combinators \(K := \lambda x y. x\) and \(S := \lambda x y z. x z (y z)\), together with the \(\lambda\)-application operator. For example, the identity \(\lambda\)-abstraction \(\lambda x. x\) is going to be \(S K K\); indeed,

\[
S K K \equiv_\beta \lambda z. K z (K z) = \lambda z. (\lambda x y. x) z (K z) \equiv_\beta \lambda z. z \equiv_\alpha \lambda x. x.
\]

Interestingly, once such a transformation is applied, one will not need the machinery of \(\lambda\)-calculus and \(\beta\)-reduction anymore. All we’ll need to do is to capture the contextual behavior of \(K\) and \(S\), which can be defined equationally very elegantly: \(K X Y = X \) and \(S X Y Z = X Z (Y Z)\), for any other \(K S\)-expressions \(X, Y, Z\).

Before we do that, we need to first discuss two other important aspects of \(\lambda\)-calculus: \(\eta\)-equivalence and extensionality.

- new info

Even though \(\lambda\)-calculus can be defined equationally and is a relatively intuitive framework, as we have noticed several times by now, substitution makes it non-trivial to implement effectively. There are several approaches in the literature addressing the subtle problem of automating substitution to avoid variable capture, all with their advantages and disadvantages. We here take a different approach. We show how \(\lambda\)-expressions

\(^1\)We use the notion of a function informally here. Functions, function application, and the notions of free and bound variables are formally discussed in Section 4.5 where we present the \(\lambda\)-calculus.
can be automatically translated into expressions over combinators, in such a way that substitution will not even be needed anymore.

A question addressed by many researchers several decades ago, still interesting today and investigated by many, is whether there is any simple equational theory that is entirely equivalent to \( \lambda \)-calculus. Since \( \lambda \)-calculus is Turing complete, such a simple theory may provide a strong foundation for computing.

*Combinatory logic* was invented by Moses Schönfinkel in 1920. The work was published in 1924 in a paper entitled “On the building blocks of mathematical logic”. Combinatory logic is a simple equational theory over two sorts, \( \text{Var} \) and \( \text{Exp} \) with \( \text{Var} < \text{Exp} \), a potentially infinite set \( x, y, \ldots \) of constants of sort \( \text{Var} \) written using lower-case letters, two constants \( K \) and \( S \) of sort \( \text{Exp} \), one application operation with the same syntax and left-associativity parsing convention as in \( \lambda \)-calculus, together with the two equations

\[
\begin{align*}
KXY &= X, \\
SXYZ &= XZ(YZ),
\end{align*}
\]

quantified universally over \( X, Y, Z \) of sort \( \text{Exp} \). The constants \( K \) and \( S \) are defined equationally in such a way to capture the intuition that they denote the combinators \( \lambda xy.x \) and \( \lambda xyz.xz(yz) \), respectively. The terms of the language, each of which denoting a function, are formed from variables and constants \( K \) and \( S \) by a single construction, function application. For example, \( S(SxKS)yS(SKxK)z \) is a well-formed term in combinatory logic, denoting some function of free variables \( x, y, \) and \( z \).

Let \( \text{CL} \) be the equational theory of combinatory logic above. Note that a function \( FV \) returning the free variables that occur in a term in combinatory logic can be defined in a trivial manner, because there are no bound variables in \( \text{CL} \). Also, note that the extensionality principle from \( \lambda \)-calculus translates unchanged to \( \text{CL} \):

\[
\text{(ext)} \quad \text{If } E_x = E'_x \text{ for some } x \notin FV(EE'), \text{ then } E = E'.
\]

Let \( \text{CL} + \text{ext} \) be \( \text{CL} \) enriched with the principle of extensionality. The following is a landmark result:

**Theorem 22.** \( \lambda + \text{ext} \) is equivalent to \( \text{CL} + \text{ext} \).

**Proof.** Let us recall what one needs to show in order for two mathematical theories to be equivalent: (1) how the syntax of one translates into the syntax of the other; and (2) that all the axioms of each of the two theories can be proved from the axioms of the other, along the corresponding translation of syntax.

Let us consider first the easy part: \( \lambda + \text{ext} \) implies \( \text{CL} + \text{ext} \). We first need to show how the syntax of \( \text{CL} + \text{ext} \) translates into that of \( \lambda + \text{ext} \). This is easy and it was already mentioned before: let \( K \) be the combinator \( \lambda xy.x \) and let \( S \) be the combinator \( \lambda xyz.xz(yz) \). We then need to show that the two equational axioms of \( \text{CL} + \text{ext} \) hold under this translation: they can be immediately proved by \( \beta \)-equivalence. We also need to show that the extensionality in \( \text{CL} + \text{ext} \) holds under the above translation: this is obvious, because it is exactly the same as the extensionality in \( \lambda + \text{ext} \).

Let us now consider the other, more difficult, implication. So we start with \( \text{CL} + \text{ext} \), where \( K \) and \( S \) have no particular meaning in \( \lambda \)-calculus, and we need to define some map that takes any \( \lambda \)-expression and translates it into an expression in \( \text{CL} \).

To perform such a transformation, let us add syntax for \( \lambda \)-abstractions to \( \text{CL} \), but without any of the equations of \( \lambda \)-calculus. This way one can write and parse \( \lambda \)-expressions, but still have no meaning for those. The following ingenious *bracket abstraction* rewriting system transforms any uninterpreted \( \lambda \)-expression into an expression using only \( K, S \), and the free variables of the original \( \lambda \)-expression:
The first rule removes all the $\lambda$-bindings, replacing them by corresponding bracket expressions. Here $\rho$ and $\rho'$ can be any expressions over $K$, $S$, variables, and the application operator, but also over the $\lambda$-abstraction operator $\lambda_{\_ \_} : \text{Var} \rightarrow \text{Exp}$. However, note that rules 2-5 systematically eliminate all the brackets. Therefore, the bracket abstraction rules above eventually transform any $\lambda$-expression into an expression over only $K$, $S$, variables, and the application operator.

The correctness of the translation of $\lambda + \text{ext}$ into $\text{CL} + \text{ext}$ via the bracket abstraction technique is rather technical: one needs to show that the translated versions of equations in $\lambda$ can be proved (by structural induction) using the machinery of $\text{CL} + \text{ext}$.

Exercise 183. (Technical) Prove the correctness of the translation of $\lambda + \text{ext}$ into $\text{CL} + \text{ext}$ above.

We do not need to understand the details of the proof of correctness in the exercise above in order to have a good intuition on why the bracket abstraction translation works. To see that, just think of the bracket abstraction as a means to associate equivalent $\lambda$-expressions to other $\lambda$-abstractions, within the framework of $\lambda$-calculus, where $K$ and $S$ are their corresponding $\lambda$-expressions. As seen above, it eventually reduces any $\lambda$-expression to one over only combinators and variables, containing no explicit $\lambda$-abstractions except those that define the combinators $K$ and $S$. To see that the bracket abstraction is correct, we can think of each bracket term $[x]E$ as the $\lambda$-expression that it was generated from, $\lambda x . E$, and then show that each rule in the bracket abstraction transformation is sound within $\lambda$-calculus. For example, rule 3 can be shown by extensionality:

$$(\lambda x . \rho \rho')(\bar{z}) \equiv_{\beta} (\rho([z/x])(\rho'([z/x]))) \equiv_{\beta} ((\lambda x . \rho z)((\lambda x . \rho' z)) \equiv_{\beta} (\lambda x y z . x y z)((\lambda x . \rho)(\lambda x . \rho') z = S((\lambda x . \rho)(\lambda x . \rho') z),$$

so by extensionality, $\lambda x . \rho \rho' \equiv_{\text{ext}} S((\lambda x . \rho)(\lambda x . \rho'))$.

This way, one can prove the soundness of each of the rules in the bracket abstraction translation. As one may expect, the tricky part is to show the completeness of the translation, that is, that everything one can do with $\lambda$-calculus and $\text{ext}$ can also do with with its subcalculus $\text{CL} + \text{ext}$. This is not hard, but rather technical.

Exercise 184. Define the bracket abstraction translation above formally in Maude. To do it, first define $\text{CL}$, then add syntax for $\lambda$-abstraction and bracket to $\text{CL}$, and then add the bracket abstraction rules as equations (which are interpreted as rewrite rules by Maude). Convince yourself that substitution is not a problem in $\text{CL}$, by giving an example of a $\lambda$-expression which would not be reducible with the definition of $\lambda$-calculus in Exercise ??. But whose translation in $\text{CL}$ can be reduced with the two equations in $\text{CL}$.

some additional notes:

In the pure system there is a small set of constants all of which are themselves pure combinators, usually the primitive combinators are named $S$, $K$ and $I$, though $I$ is in fact definable in terms of $S$ and $K$.  

314
The intended meaning of these constants can be described simply by showing a corresponding transformation or reduction which the constant may be thought of as effecting.

The transformations are as follows: \( I \times x = x \) \( K \times x \times y = x \) \( S \times f \times g \times x = (f \times x)(g \times x) \) This set of combinators has the property of combinatory completeness which means that for any function definition of the form: \( f \times x = \) combinatory expression involving \( x \)

There exists an expression \( E \) in \( S,K \) and \( I \) such that: \( f = E \)

This property establishes the close relationship between Pure Combinatory Logic and the Pure lambda Calculus, in which a special notation for functional abstraction is available. It shows that the notation for functional abstraction, though a great convenience, adds no expressiveness to the language.

As well as having combinatorial completeness, Pure Combinatory Logic is able to express all effectively computable functions over natural numbers appropriately represented as combinators.


This is the first published work in the field which came later to be known as combinatory logic.

Schönfinkel shows how the use of bound variables in logic can be dispensed with. The use of higher order functions makes possible the reduction of logic to a language consisting of one constructor (the application of a function to an argument) and three primitive constants \( U, C \) (now usually called \( K \)) and \( S \). A function is termed “higher order” if it will accept a function as an argument, or return one as its result. \( U, C, \) and \( S \) are all higher order functions.

The meaning of these primitives may be understood through the following informal definitions:

\[ C \times x \times y = (C \times x)(y) = x \]
\( C \) is a function which given any value \( x \), returns the constant \( x \) valued function.

\[ S \times f \times g \times x = ((S \times f)(g))(x) = (f \times x)(g \times x) \]
\( S \) is a function which combines two functions, say \( f \) and \( g \), supplied as successive arguments. The resulting function, given a value \( x \), returns the value obtained by applying the value of \( f \) at \( x \) to the value of \( g \) at \( x \). \( U \times P \times Q = (U \times P)(Q) = \) forall \( x. \) not( \( P \times x \) and \( Q \times x \) ) \( U \) is a generalised scheffer stroke. It should be thought of as applying to two predicates and returns the universal quantification of the negated conjunction of the two predicates.

These combinators are sufficient to enable arbitrary first order predicates to be expressed without the use of bound variables, which appear in first order logic whenever a quantifier is used. This can be demonstrated most straightforwardly using a simple algorithm which converts lambda-expressions to combinators. They are not limited to first-order predicates, but without some constraints (equivalent to those found in first order logic, or more liberally and appropriately to those in Church’s Simple Theory of Types [Church40]) the logic which results from the use of these combinators is at risk of proving inconsistent. Combinatory logic has traditionally tried to avoid type constraints and has therefore been dogged with difficulties in achieving strength and consistency.

Schönfinkel’s paper remains an accessible introduction to combinatory logic which makes clear the original motivation for this innovation. This original motivation was vigorously pursued later by H.B.Curry and his collaborators. The results of Curry’s programme were published in Combinatory Logic, Vols 1 & 2.

While Schönfinkel was successful in showing that combinators could provide plausible notation, the semantic and proof theoretic aspects were more difficult to put in place. The difficulty is more pronounced if it hoped to use combinatory logic as a foundation for mathematics, than it is for the use of combinators in less demanding roles (e.g. for programming languages).

The Pure combinatory logic may be thought of as Schönfinkels system with the logical operator ”\( U \)” omitted (though the presentation may vary). Illative combinatory logics are formed by adding additional logical primitives to the pure combinators. There are many illative combinatory logics, which vary among themselves in substance as well as in presentation.
Pure combinatory logic is so closely related to Church’s lambda-calculus that it is best studied alongside the lambda-calculus, for which the most comprehensive modern text is probably The Lambda Calculus. A very popular and entertaining introduction to the pure combinators may be found in To Mock a Mockingbird.

Research on illative combinatory logics has been continued by Curry’s students, Roger Hindley and Jonathan Seldin, and by Martin Bunder. Hindley, Lercher and Seldin published An Introduction to Combinatory Logic, an excellent and readable short introduction to the subject, now superseded by the more comprehensive An Introduction to Combinators and the Lambda-Calculus.

Recently Randall Holmes has developed logical systems based on combinatory logic and the lambda calculus using methods derived from Quine’s formulations of set theory.

The attempt to make a foundation for mathematics using the illative combinatory logic were blighted by Curry’s paradox and semantic opacity. While systems avoiding this paradox have been devised, they are typically weak, and lack the kind of convincing semantic intuition found in first order set theory.

4.2 Turing Machines

4.2.1 Notes

Encodings of Turing computability into equational deduction and/or rewriting are well-known. For example, Bergstra and Tucker [7] (1995) and Baader and Nipkow [11] (1998) give encodings that have properties similar to ours in this section, in the sense that their resulting equational specifications, if regarded as term rewrite systems, are confluent and terminate whenever the original computation terminates. It seems, however, that due to the particular two-infinite-end-tape variant of Turing machine chosen in this chapter, our encoding is somewhat simpler. The ideas underlying our encodings presented in this section were first published by the author in 2006 [65].

As mentioned at the beginning of this section, our purpose here was not to address the wonderful theoretical implications of Turing machines and computability, but rather to regard the Turing machine through the programming language lens and to show how we can give it a rewriting semantics just like for all the languages discussed in this book.
Instructions are labeled tales, such as:

1: ◦
   •◦◦
2: ◦•
   ◦◦
3: ••◦
   ••

Programs are label sequences, such as:

3 2 3 1

Figure 4.1: Post correspondence problem as a programming language

4.3 Post Correspondence

Turing machines allow us to formally state and prove that certain problems are undecidable, that is, that they cannot be solved by computers, no matter what programming language is being used. The canonical undecidable problem in the context of Turing machines is the halting problem: given a turing machine \( M \) and an input \( i \), does \( M \) halt on \( i \)? Because of its reflective nature, the halting problem is somehow complicated to use in practice. In this section we discuss another canonical undecidable problem, the Post correspondence problem, showing how it can be reduced to a step-for-step equivalent rewrite theory. The reason for which we discuss this problem in this chapter dedicated to idealized programming languages, is because we can indeed regard it as an overly simplified programming language, whose basic instructions are two-dimensional tales and whose programs are sequences of such tales, as shown in Figure ??.
4.4 Counter Machines
4.5 Lambda Calculus

Lambda calculus was introduced in 1930s, as a mathematical theory together with a proof calculus aiming at capturing foundationally the important notions of function and function application. Those years were marked by several paradoxes in mathematics and logics. The original motivation of \( \lambda \)-calculus was to provide a foundation of logics and mathematics. Whether \( \lambda \)-calculus indeed provides a strong foundation of mathematics is still open; what is certain is that \( \lambda \)-calculus was and still is a quite successful theory of computation. Today, more than 70 years after its birth, \( \lambda \)-calculus and its afferent subjects still fascinates computer scientists, logicians, mathematicians and, certainly, philosophers.

\( \lambda \)-Calculus is a convenient framework to describe and explain many programming language concepts. It formalizes the informal notion of “expression that can be evaluated” as a \( \lambda \)-term, or \( \lambda \)-expression. More precisely, \( \lambda \)-calculus consists of:

- **Syntax** - used to express \( \lambda \)-terms, or \( \lambda \)-expressions;
- **Proof system** - used to prove \( \lambda \)-expressions equal;
- **Reduction** - used to reduce \( \lambda \)-expressions to equivalent ones.

We will show how \( \lambda \)-calculus can be formalized as an *equational theory* in equational logic (Section 2.4), provided that an important mechanism of *substitution* is available. That means that its syntax can be defined as an algebraic signature; its proof system becomes a special case of equational deduction; and its reduction becomes a special case of rewriting (when certain equations are regarded as rewrite rules). These results support the overall theme underlying this book, that equational logic and rewriting are a strong foundation for describing and explaining programming language concepts. Even though \( \lambda \)-calculus is a special equational theory, it has the merit that it is powerful enough to express most programming language concepts quite naturally. Equational logic gives us, in some sense, “too much freedom” in how to define concepts; its constraints and intuitions are not restrictive enough to impose an immediate mapping of programming language concepts into it. As extensively illustrated in Chapter 3, due to their generality, equational and rewrite logics need appropriate methodologies to define languages, while \( \lambda \)-calculus, due to its particularity, gives an immediate means to syntactically encode programming languages and hereby give them a semantics.

As discussed in Section 4.5.6, many programming language concepts, and even entire programming languages, translate relatively naturally into \( \lambda \)-calculus concepts or into \( \lambda \)-expressions. That includes the various types of values and numbers, data structures, recursion, as well as complex language constructs.

### 4.5.1 Syntax

Syntactically, \( \lambda \)-calculus consists of two syntactic constructs, \( \lambda \)-abstraction and \( \lambda \)-application, and a given infinite set of variables. Let us assume an infinite set of variables, or names, \( \text{Var} \). Then the syntax of \( \lambda \)-expressions is given by the following grammar (in BNF notation):

\[
\begin{align*}
\text{Exp} & ::= \text{Var} \quad \text{— variables are } \lambda \text{-expressions} \\
& \mid \lambda \text{Var}.\text{Exp} \quad \text{— } \lambda \text{-abstraction, or nameless function} \\
& \mid \text{Exp} \text{ Exp} \quad \text{— } \lambda \text{-application}
\end{align*}
\]

We will use lower letters \( x, y, z \), etc., to refer to variables, and capital letters \( E, E', E_1, E_2 \), etc., to refer to \( \lambda \)-expressions. The following are therefore examples of \( \lambda \)-expressions:
\[ \lambda x. x \]
\[ \lambda x. xx \]
\[ \lambda x. (f x)(g x) \]
\[ (\lambda x. f x) x \]

\( \lambda \)-Expressions of the form \( \lambda x. E \) are called \( \lambda \)-abstractions, and those of the form \( E_1 E_2 \) are called \( \lambda \)-applications. The former captures the intuition of nameless functions, while the latter that of function applications. To avoid parentheses, we assume that \( \lambda \)-application is left associative and binds tighter than \( \lambda \)-abstraction; for example, \( \lambda x. \lambda y. \lambda z.xyz \) is the same as \( \lambda x. \lambda y. \lambda z.(xy)z \).

### 4.5.2 Free and Bound Variables

Variable occurrences in \( \lambda \)-expressions can be either free or bound. Given a \( \lambda \)-abstraction \( \lambda x. E \), also called a binding, then the variable \( x \) is said to be declared by the \( \lambda \)-abstraction, or that \( \lambda x. E \) binds \( x \) in \( E \); also, \( E \) is called the scope of the binding. Formally, we define the set \( FV(E) \) of free variables of \( E \) as follows:

- \( FV(x) = \{ x \} \),
- \( FV(E_1 E_2) = FV(E_1) \cup FV(E_2) \), and
- \( FV(\lambda x. E) = FV(E) \setminus \{ x \} \).

Consider the three underlined occurrences of \( x \) in the \( \lambda \)-expression \( (\lambda x. \lambda y. yy)x \). The first is called a binding occurrence of \( x \), the second a bound occurrence of \( x \) (this occurrence of \( x \) is bound to the binding occurrence of \( x \)), and the third a free occurrence of \( x \). Expressions \( E \) with \( FV(E) = \emptyset \) are called closed or combiners.

### 4.5.3 Substitution

Evaluation of \( \lambda \)-expressions is by substitution. That means that the \( \lambda \)-expression that is passed to a \( \lambda \)-abstraction is copied as is at all the bound occurrences of the binding variable. This will be formally defined later. Let us now formalize and discuss the important notion of substitution. Intuitively, \( E[E'/x] \) represents the \( \lambda \)-expression obtained from \( E \) by replacing each free occurrence of \( x \) by \( E' \). For notational simplicity, we assume that, syntactically, substitution binds tighter than any \( \lambda \)-calculus construct; as usual, parentheses can be used to enforce desired grouping. Formally, substitution can be defined as follows:

- \( y[E'/x] = \begin{cases} E' & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \)
- \( (E_1 E_2)[E'/x] = E_1[E'/x] E_2[E'/x] \)
- \( (\lambda x. E)[E'/x] = \lambda x. E \)

The tricky part is to define substitution on \( \lambda \)-abstractions of the form \( (\lambda y. E)[E'/x] \), where \( y \neq x \). That is because \( E' \) may contain free occurrences of \( y \); these occurrences of \( y \) would become bound by the binding variable \( y \) if one simply defined this substitution as \( \lambda y. (E[E'/x]) \) (and if \( E \) had any free occurrences of \( x \)), thus violating the intuitive meaning of binding. This phenomenon is called variable capturing. Consider, for example, the substitution \( (\lambda y. x)[yy/x] \); if one applies the substitution blindly then one gets \( \lambda y. yy \), which is most likely not what one meant (since \( \lambda y. x \) is by all means equivalent to \( \lambda z. x \)—this equivalence will be formalized shortly—while \( \lambda y. yy \) is not equivalent to \( \lambda z. yy \)). There are at least three approaches in the literature to deal with this delicate issue:
1. Define \((\lambda y. E)[E'/x]\) as \(\lambda y. (E[E'/x])\), but pay special attention whenever substitution is used to add sufficient conditions to assure that \(y\) is not free in \(E'\). This approach simplifies the definition of substitution, but complicates the presentation of \(\lambda\)-calculus by having to mention obvious additional hypotheses all the time a substitution is invoked.

2. Define substitution as a partial operation: \((\lambda y. E)[E'/x]\) is defined and equal to \(\lambda y. (E[E'/x])\) if and only if \(y \not\in \text{FV}(E')\). This may seem like the right approach, but unfortunately is also problematic, because the entire equational definition of \(\lambda\)-calculus would then become partial, which has serious technical implications w.r.t. mechanizing equational deduction (or the process of proving \(\lambda\)-expressions equivalent) and rewriting (or reduction);

3. Define substitution as a total operation, but apply a renaming of \(y\) in \(\lambda y. E\) to some variable that does not occur in \(E\) or \(E'\) in case \(y \in \text{FV}(E')\) (this renaming is called \(\alpha\)-conversion and will be defined formally shortly). This approach slightly complicates the definition of substitution, but simplifies the presentation of many results later on. It is also useful when one wants to mechanize \(\lambda\)-calculus, because it provides an algorithmic way to avoid variable capturing:

\[
(\lambda y. E)[E'/x] = \begin{cases} 
\lambda y. (E[E'/x]) & \text{if } y \not\in \text{FV}(E') \\
\lambda z. ((E[z/y])[E'/x]) & \text{if } y \in \text{FV}(E') 
\end{cases}
\]

where \(z\) is a new variable that does not occur in \(E\) or \(E'\). Note that the the requirement “\(z\) does not occur in \(E\) or \(E'\)” is stronger than necessary, but easier to state that way.

All three approaches above have their advantages and disadvantages, with no absolute best approach. Sections 4.5.9 and 4.5.10 discuss alternative approaches to define \(\lambda\)-calculus, where the explicit lambda bindings and the implicit variable capturing problem are eliminated by design.

### 4.5.4 Alpha Conversion

In mathematics, functions that differ only in the name of their variables are equal. For example, the functions \(f\) and \(g\) defined (on the same domain) as \(f(x) = x\) and \(g(y) = y\) are considered identical. However, with the machinery developed so far, there is no way to show that the \(\lambda\)-expressions \(\lambda x. x\) and \(\lambda y. y\) are equal. It is common in the development of mathematical theories to add desirable but unprovable properties as axioms. The following is the first meaningful equational axiom of \(\lambda\)-calculus, known under the name of \(\alpha\)-conversion:

\[
\lambda x. E = \lambda z. (E[z/x])\]

for any variable \(z\) that does not occur in \(E\) \((\alpha)\)

The requirement on \(z\) not occurring in \(E\) is again stronger than necessary, but it is easier to state.

Using the equation above, one has now the possibility to formally prove \(\lambda\)-expressions equivalent. To capture this provability relation formally, we let \(E \equiv_\alpha E'\) denote the fact that the equation \(E = E'\) can be proved using standard equational deduction from the equational axioms above \((\alpha)\) plus those for substitution.

### 4.5.5 Beta Equivalence and Beta Reduction

We now define another important equation of \(\lambda\)-calculus, known under the name of \(\beta\)-equivalence:

\[
(\lambda x. E)E' = E[E'/x] \tag{\beta}
\]

322
The equation (β) tells us how λ-abstractions are applied. Essentially, the argument λ-expression passed to a λ-abstraction is copied at every free occurrence of the variable bound by the λ-abstraction within its scope.

We let \( E \equiv_\beta E' \) denote the fact that the equation \( E = E' \) can be proved using standard equational deduction from the equational axioms above: (a), (β), plus those for substitution. For example \( (\lambda f. f x)(\lambda y) y \equiv_\beta x \), because one can first deduce that \( (\lambda f. f x)(\lambda y) y = (\lambda y) x \) by (β) and then that \( (\lambda y) x = x \) also by (β); the rest follows by the transitivity rule of equational deduction.

When the equation (β) is applied only from left to write, that is, as a rewrite rule, it is called β-reduction. We let \( \Rightarrow_\beta \) denote the corresponding rewriting relation on λ-expressions. To be more precise, the relation \( \Rightarrow_\beta \) is defined on α-equivalence classes of λ-expressions; in other words, \( \Rightarrow_\beta \) applies modulo α-equivalence.

Given a λ-expression \( E \), one can always apply α-conversion on \( E \) to rename its binding variables so that all these variables have different names and they do not occur in \( \text{FV}(E) \). If that is the case, then note that variable capturing cannot occur when applying a β-reduction step. In particular, that means that one can follow the first, i.e., the simplest approach of the three discussed previously in Section 4.5.3 to define or implement substitution. In other words, if one renames the binding variables each time before applying a β-reduction, then one does not need to rename binding variables during substitution. This is so convenient in the theoretical developments of λ-calculus, that most works on this subject adopt the following convention: all the binding variables occurring in any given λ-expression at any given moment are assumed to be different; moreover, it is assumed that a variable cannot occur both free and bound in any λ-expression. If a λ-expression does not satisfy this, then one can apply a certain number of α-conversions and eventually transform it into an α-equivalent one that does satisfy it. Clearly, this process of renaming potentially all the binding variables before applying any β-reduction step may be computationally quite expensive. In a more familiar setting, it is like traversing and changing the names of all the variables in a program at each execution step! There are techniques aiming at minimizing the amount of work to be performed in order to avoid variable capturing. All these techniques, however, incur certain overheads.

One should not get tricked by thinking that one renaming of the binding variables, at the beginning of the reduction process, should be sufficient. It is sufficient for just one step of β-reduction, but not for more. Consider, for example, the closed λ-expression, or the combinator, \( (\lambda z. z z)(\lambda x. \lambda y. x y) \). It has three binding variables, all distinct, so their renaming makes no difference. However, if one applies substitution in β-reductions blindly then one quickly ends up capturing the variable \( y \):

\[
(\lambda z. z z)(\lambda x. \lambda y. x y) \Rightarrow_\beta (\lambda x. \lambda y. x y)(\lambda x. \lambda y. x y) \Rightarrow_\beta \lambda y. (\lambda x. \lambda y. x y)(\lambda x. \lambda y. x y) y \Rightarrow_\beta \lambda y. \lambda y. y y
\]

Section 4.5.9 will discuss an ingenious technique to transform λ-calculus into combinatory logic which, surprisingly, eliminates the need for substitutions entirely.

Consider the λ-expression \( \lambda f. (\lambda x. f x y) y \). There are two different ways to apply β-reduction on it:

1. \( (\lambda f. (\lambda x. f x y)) y \Rightarrow_\beta (\lambda f. f y) y \), and

2. \( (\lambda f. (\lambda x. f x y)) y \Rightarrow_\beta (\lambda x. g x) y \).

Nevertheless, both resulting λ-expressions above can be further reduced to \( g y \) by applying β-reduction. This brings us to one of the most notorious results in λ-calculus (\( \Rightarrow_\beta^* \) is the reflexive and transitive closure of \( \Rightarrow_\beta \)):

**Theorem 23.** \( \Rightarrow_\beta \) is confluent. That means that for any λ-expression \( E \), if \( E \Rightarrow_\beta^* E_1 \) and \( E \Rightarrow_\beta^* E_2 \) then there is some λ-expression \( E' \) such that \( E_1 \Rightarrow_\beta^* E' \) and \( E_2 \Rightarrow_\beta^* E' \). All this is, of course, modulo α-conversion.

The confluence theorem above says that it essentially does not matter how the β-reductions are applied on a given λ-expression. A λ-expression is called a β-normal form if no β-reduction can be applied on it.
A λ-expression $E$ is said to admit a β-normal form if and only if there is some β-normal form $E'$ such that $E \Rightarrow^*_\beta E'$. The confluence theorem implies that if a λ-expression admits a β-normal form then that β-normal form is unique modulo α-conversion.

Note, however, that there are λ-expressions which admit no β-normal form. Consider, for example, the λ-expression $(λx.x) (λx.x)$, known as the divergent combinator $\omega$. It is easy to see that $\omega \Rightarrow_\beta \omega$ and that’s the only β-reduction that can apply on $\omega$, so it has no β-normal form.

### 4.5.6 Lambda Calculus as a Programming Language

In contrast to Turing machines, which are rather low-level computational devices, λ-calculus is closer in spirit to higher-level programming languages, particularly to functional programming languages. In this section we show how several common programming language features, including particular language constructs, data-types, and recursion, can be naturally represented as λ-expressions. We say that such language features are derived (from λ-calculus), or syntactic sugar (for equivalent but harder to read λ-expressions). All these suggest that λ-calculus is rich and intuitive enough to safely be regarded as a programming language itself.

#### Anonymous Functions and Let Binders

Most functional programming languages, and also some other languages which are not necessarily considered functional, provide anonymous, or nameless functions, which are functions that are not bound to any particular name. For example, the anonymous successor function is written as `fn x => x+1` in the Standard ML functional language. Anonymous functions can still be applied to arguments, for example `(fn x => x+1) 2` evaluates to 3 in Standard ML, and they can typically be passed as arguments to or returned as results by other functions. Almost all languages supporting anonymous functions have a different syntax for them. Nevertheless, anonymous functions are nothing but λ-abstractions. For example, the successor function above can be regarded as the λ-abstraction $λx.(x+1)$, where 1 and + are either builtins or can be also encoded as λ-expressions (these are discussed in the sequel).

Another common construct in programming languages is the `let` binder, which allows to bind expressions to names and then use those names as replacements for their corresponding expressions. For example,

$\text{let } x1 = E1 \text{ and } x2 = E2 \text{ and} \ldots \text{and } xn = En \text{ in } E$

first binds the expressions $E1, E2, \ldots, En$ to the names $x1, x2, \ldots, xn$, respectively, and then $E$ is evaluated. It is not hard to see that this construct is equivalent to the λ-expression $(λx1.λx2.\ldots.λxn.E) \ E1 \ E2 \ldots \ En$.

#### Currying

Recall from Section 2.1.1 that there is a bijection between $[A \times B \rightarrow C]$ and $[A \rightarrow [B \rightarrow C]]$, where $[X \rightarrow Y]$ represents the set of functions of domain $X$ and codomain $Y$. Indeed, any function $f : A \times B \rightarrow C$ can be regarded as a function $g : A \rightarrow [B \rightarrow C]$, where for any $a \in A$, $g(a)$ is defined as the function $h_a : B \rightarrow C$ with $h_a(b) = c$ if and only if $f(a, b) = c$. Similarly, any function $g : A \rightarrow [B \rightarrow C]$ can be regarded as a function $f : A \times B \rightarrow C$, where $f(a, b) = g(a)(b)$.

This observation led to the important concept called currying, which allows us to eliminate functions with multiple arguments from the core of a language, replacing them systematically by functions admitting only one argument as above. Thus, functions with multiple arguments are just syntactic sugar.

From now on we may write λ-expressions of the form $\lambda x_1 x_2 \cdots. E$ as shorthands for their uncurried versions $\lambda x.\lambda y.\lambda z.\cdots. E$. With this convention, λ-calculus therefore admits multiple-argument λ-abstractions.
Note, however, that unlike in many familiar languages, curried functions can be applied on fewer arguments. For example, \((\lambda xyz. E) E'\) \(\beta\)-reduces to \(\lambda yz.(E[E'/x])\). Also, since \(\lambda\)-application was defined to be left-associative, \((\lambda xyz. E) E_1 E_2\) \(\beta\)-reduces to \(\lambda z.(E_1/E_1)[E_2/y]\).

Most functional languages today support curried functions. The advantage of currying is that one only needs to focus on defining the meaning or on implementing effectively functions of one argument.

**Church Booleans**

Booleans are perhaps the simplest data-type that one would like to have in a programming language. \(\lambda\)-calculus so far provides no explicit support for Booleans or conditionals. We next show that \(\lambda\)-calculus provides *implicit* support for Booleans. In other words, the machinery of \(\lambda\)-calculus is powerful enough to simulate Booleans and what one would normally want to do with them in a programming language. What we discuss next is therefore a *methodology* to program with Booleans in \(\lambda\)-calculus.

The idea is to regard a Boolean through a behavioral prism: with a Boolean, one can always choose one of any two objects—if true then the first, if false then the second. In other words, one can identify a Boolean \(b\) with a universally quantified conditional “for any \(x\) and \(y\), if \(b\) then \(x\) else \(y\)”. With this behavior of Booleans in mind, one can now relatively easily translate Booleans and Boolean operations in \(\lambda\)-calculus:

\[
\text{true} := \lambda xy.x \\
\text{false} := \lambda xy.y \\
\text{if-then-else} := \lambda xyz.xyz \\
\text{and} := \lambda xy.(x y \text{false})
\]

Other Boolean operations can be defined in a similar style (see Exercise 198).

This encoding for Booleans is known under the name of Church Booleans. We can use \(\beta\)-reduction to show, for example, that \(\text{and true false} \Rightarrow_{\beta} \text{false}\), that is, \(\text{true false} \equiv_{\beta} \text{false}\). We can show relatively easily that the Church Booleans have all the desired properties of Booleans. Let us, for example, show the associativity of \(\text{and}\). First, note the obvious equivalences:

\[
\text{and} (\text{and } x y) z \equiv_{\beta} x y \text{false} z \text{false} \\
\text{and} x(\text{and } y z) \equiv_{\beta} x (y z \text{false}) \text{false}
\]

These equivalences tell us that we cannot expect the properties of Booleans to hold for any \(\lambda\)-expressions. Therefore, in order to complete the proof of associativity of \(\text{and}\), we need to make further assumptions regarding the Booleanity of \(x, y, z\). If \(x\) is true, that is \(\lambda xy.x\), then both right-hand-side \(\lambda\)-expressions above reduce to \(y z \text{false}\). If \(x\) is false, that is \(\lambda xy.y\), then the first reduces to \(\text{false} z \text{false}\) which further reduces to \(\text{false}\), while the second reduces to \(\text{false}\) in one step. We can similarly prove the desired properties of all the Boolean operators (Exercise 199), relying only on the generic \(\beta\)-equivalence of \(\lambda\)-calculus.

We may often introduce definitions such as the above for the Church Booleans, using the symbol \(\equiv\). Note that this is not a meta binding constructor on top of \(\lambda\) calculus. It is just a way for us to avoid repeating certain frequent \(\lambda\)-expressions; one can therefore regard them as macros. Anyway, they admit a simple translation into standard \(\lambda\)-calculus, using the usual convention for translating bindings. Therefore, one can regard the \(\lambda\)-expression “and true false” as syntactic sugar for

\[
(\lambda \text{and}. \lambda \text{true}. \lambda \text{false}. \text{and true false}) ((\lambda \text{false}. \lambda xy. \text{false}) (\lambda xy.y)) (\lambda xy.x) (\lambda xy.y)
\]
Pairs

\(\lambda\)-calculus can also naturally encode data-structures of interest in most programming languages. The idea is that \(\lambda\)-abstractions, by their structure, can store useful information. Let us, for example, consider pairs as special cases of records. Like Booleans, pairs can also be regarded behaviorally: a pair is a black-box that can store any two expressions and then allow one to retrieve those through appropriate projections. Formally, we would like to define \(\lambda\)-expressions \texttt{pair}, \texttt{1st} and \texttt{2nd} in such a way that for any other \(\lambda\)-expressions \(E\) and \(E'\), it is the case that \(\texttt{1st}(\texttt{pair} E E')\) and \(\texttt{2nd}(\texttt{pair} E E')\) are \(\beta\)-equivalent to \(E\) and \(E'\), respectively. Fortunately, these can be defined quite easily:

\[
\begin{align*}
\texttt{pair} & := \lambda xy. b xy, \\
\texttt{1st} & := \lambda p. p \texttt{true}, \text{ and} \\
\texttt{2nd} & := \lambda p. p \texttt{false}.
\end{align*}
\]

The idea is therefore that \(\texttt{pair} E E'\) \(\beta\)-reduces to the \(\lambda\)-expression \(\lambda b. b E E'\), which freezes \(E\) and \(E'\) inside a \(\lambda\)-abstraction, together with a handle, \(b\), which is expected to be a Church Boolean, to unfreeze them later. Indeed, the first projection, \(\texttt{1st}\), takes a pair and applies it to \texttt{true} hereby unfreezing its first component, while the second projection applies it to \texttt{false} to unfreeze its second component.

Church Numerals

Numbers and the usual operations on them can also be defined as \(\lambda\)-expressions. The basic idea is to regard a natural number \(n\) as a \(\lambda\)-expression that has the potential to apply a given operation \(n\) times on a given starting \(\lambda\)-expression. Therefore, \(\lambda\)-numerals, also called \textit{Church numerals}, take two arguments, \textit{what to do} and \textit{what to start with}, and apply the first as many times as the intended numeral on the second. Intuitively, if the action was \textit{successor} and the starting expression was \textit{zero}, then one would get the usual numerals. Formally, we define numerals as follows:

\[
\begin{align*}
0 & := \lambda sz. z \\
1 & := \lambda sz. s z \\
2 & := \lambda sz. s (s z) \\
3 & := \lambda sz. s (s (s z)) \\
& \ldots
\end{align*}
\]

With this intuition for numerals in mind, we can now easily define a successor operation on numerals:

\[
\texttt{succ} := \lambda nsz. n s (s z)
\]

The above says that for a given numeral \(n\), its successor \(\texttt{succ} n\) is the numeral that applies the given operation \(s\) for \(n\) times starting with \(s z\). There may be several equivalent ways to define the same intended meaning. For example, one can also define the successor operation by applying the operation \(s\) only once, but on the expression \(n s z\); therefore, we can define

\[
\texttt{succ'} := \lambda nsz. s (n s z)
\]

326
One may, of course, want to show that \( succ \) and \( succ' \) are equal. However, they are not equal as \( \lambda \)-expressions. To see it, one can apply both of them on the \( \lambda \)-expression \( \lambda x y. x \): one gets after \( \beta \)-reduction \( \lambda s z. s \) and, respectively, \( \lambda s z. s s \). However, they are equal when applied on Church numerals (see Exercise 200).

One can also define addition as a \( \lambda \)-abstraction, e.g., as follows:

\[
\text{plus} := \lambda m n s z. m (s (n s z))
\]

One of the most natural questions that one can and should ask when one is exposed to a new model of natural numbers, is whether it satisfies the Peano axioms. In our case, this translates to whether the following properties hold for all Church numerals \( n \) and \( m \) (see Exercise 201):

\[
\text{plus} 0 \equiv m, \text{ and } \\
\text{plus} (\text{succ} n) m \equiv \text{succ} (\text{plus} n m).
\]

Church numerals in combination with pairs allow us to define certain recursive behaviors. Let us next define a more interesting function on Church numerals, namely one that calculates Fibonacci numbers. Specifically, we want to define a \( \lambda \)-expression \( \text{fibo} \) with the property that \( \text{fibo} \ n \beta \)-reduces to the \( n \)th Fibonacci number. Recall that Fibonacci numbers are defined recursively as \( f_0 = 0, f_1 = 1, \) and \( f_n = f_{n-1} + f_{n-2} \) for all \( n \geq 2 \). The idea is to define a two-number window that slides through the sequence of Fibonacci numbers until it reaches the desired number. The window is defined as a pair and the sliding by moving the second element in the pair on the first position and placing the next Fibonacci number as the second element. The shifting operation needs to be applied as many times as the index of the desired Fibonacci number:

\[
\text{start} := \text{pair } 0 1, \\
\text{step} := \lambda p. \text{pair} (2 \text{nd } p) (\text{plus} 1 \text{st } p (2 \text{nd } p)), \\
\text{fibo} := \lambda n. \text{1st} (n \text{ step start}).
\]

We will shortly discuss a technique to support recursive definitions of functions in a general way, not only on Church numerals.

Another use of the technique above is in defining the predecessor operation on Church numerals:

\[
\text{start} := \text{pair } 0 1, \\
\text{step} := \lambda p. \text{pair} (2 \text{nd } p) (\text{plus} 1 \text{st } p (2 \text{nd } p)), \\
\text{pred} := \lambda n. \text{1st} (n \text{ step start}).
\]

Note that \( \text{pred} 0 \equiv 0 \), which is a slight violation of the usual properties of the predecessor operation on integers. The above definition of predecessor is computationally very inefficient. Unfortunately, there does not seem to be any better way to define this operation on Church numerals.

Subtraction can now be defined easily:

\[
\text{sub} := \lambda m n. n \text{ pred } m.
\]

Note, again, that negative numbers are collapsed to \( 0 \).

Let us next see how relational operators can be defined on Church numerals. These are useful to write many meaningful programs. We first define a helping operation, to test whether a number is zero:

\[
\text{327}
\]
zero? := λn. n (and false) true.

Now the “less than or equal to” (leq), the “larger than or equal to” (geq), and the “equal to” (equal) can be defined as follows:

leq := λmn. zero? (sub m n),
geq := λmn. zero? (sub n m),
equal := λmn. and (leq m n) (geq m n).

Adding Built-ins

As shown above, λ-calculus is powerful enough to encode many data-structures and data-types of interest. However, as it is the case with many other pure programming paradigms, in order for λ-calculus to be usable as a reasonably efficient programming language, it needs to provide built-ins comprising efficient implementations for common data-types and operations on them.

We here only discuss the addition of built-in integers to λ-calculus; the addition of other data-types is similar. We say that the new λ-calculus obtained this way is enriched. Depending on how the underlying substitution operation and how the built-in operations are defined, enriching λ-calculus with built-in integers can take very little effort: syntactically, we may only need to add integers as λ-expressions, and semantically, we may only need to add the rules defining the built-in operations to the language, in addition to the already existing β-reduction rule. Specifically, assume a substitution operation that works over any syntactic extension of λ-calculus, essentially applying homomorphically through all non-binding language constructs, like in Exercise [196]. Also, assume a syntactic category Int for integer values, e.g., 5, 7, etc., and operations on them, e.g., +, together with a rewrite relation capable of reducing Int expressions, e.g., 5+7 ⇒ 12.

There are at least two ways to enrich λ-calculus with integers: the simplest is to collapse the Int and Exp syntactic categories; another is to make Int a syntactic subcategory of Exp, but then all the Int operations need to be overloaded to take and yield Exp. In both cases, the integer values become particular expressions, and the integer reduction rules harmoniously co-exist with β-reduction. For example, 5+7 still reduces to 12. But what is more interesting is that λx.7+5 also makes sense and reduces to λx.12 (without applying any β-reduction step, but only the reduction that Int provides). Moreover, (λyx.7+y) 5 first β-reduces to λx.7+5 and then Int-reduces to λx.12. The above β-reduction works because integer values are regarded as constants, that is, operations with no arguments, so the generic substitution (homomorphically) applies on them, too, keeping them unchanged, as expected: I[E'/x] = I, for any integer I.

Note that one can now write λ-expressions that are not well formed, such as the λ application of one integer to another: 7 5. It would be the task of a type checker to catch such kind of errors. We here focus only on the evaluation, or reduction, mechanism of the enriched calculus, assuming that programs are well-formed. Ill-formed λ-expressions may get stuck when β-reduced.

4.5.7 Fixed-Points and Recursion

Recursion almost always turns out to be a subtle topic in foundational approaches to programming languages. We have already seen the divergent combinator

omega := (λx. x x) (λx. x x),

which has the property that omega ⇒_β omega, that is, it leads to an infinite recursion. While omega has a recursive behavior, it does not give us a principled way to define recursion in λ-calculus.
But what is recursion? Or to be more precise, what is a recursive function? Let us examine the definition of a factorial function, in some conventional programming language, that we would like to be recursive:

```plaintext
function f(x) {
    if x == 0 then 1 else x * f(x - 1)
}
```

In a functional language that is closer in spirit to $\lambda$-calculus the definition of factorial would be:

```plaintext
letrec
    f(x) = if x == 0 then 1 else x * f(x - 1)
in f(3).
```

Note that the `letrec` binding is necessary in the above definition. If we used `let` instead, then according to the syntactic sugar transformation of functional bindings into $\lambda$-calculus, the above would be equivalent to

```plaintext
(\lambda f . f 3)
(\lambda x . if x == 0 then 1 else x * f(x - 1)),
```

so the underlined $f$ is free rather than bound to $\lambda f$, as expected. A functional programming language would report an error here.

The foundational question regarding recursion in $\lambda$-calculus is therefore the following: how can we define a $\lambda$-abstraction

```plaintext
f := \begin{exp} \ldots f \ldots \end{exp},
```

that is, one in which the $\lambda$-expression refers to itself in its scope? Let us put the problem in a different light. Consider instead the well-formed well-behaved $\lambda$-expression

```plaintext
F := \lambda f . \begin{exp} \ldots f \ldots \end{exp},
```

that is, one which takes any $\lambda$-expression, in particular a $\lambda$-abstraction, and plugs it at the right place into the scope of the $\lambda$-expression that we want to define recursively. The question now becomes the following one: can we find a fixed point $f$ of $F$, that is, a $\lambda$-expression $f$ with the property that

```plaintext
F f \equiv_{\beta} f?
```

Interestingly, $\lambda$-calculus has the following notorious and perhaps surprising result:

**Theorem 24. (Fixed-point theorem of untyped $\lambda$-calculus)** For any $\lambda$-expression $F$, there is some $\lambda$-expression $X$ such that $FX \equiv_{\beta} X$.

**Proof.** One such $X$ is the $\lambda$-expression $(\lambda x. F (x x)) (\lambda x. F (x x))$. Indeed,

\[
X = (\lambda x. F (x x)) (\lambda x. F (x x)) \\
\equiv_{\beta} F ((\lambda x. F (x x)) (\lambda x. F (x x))) \\
= FX.
\]

Therefore, $X$ is a fixed-point of $F$. 

The proof of the fixed-point theorem above suggests defining the following famous fixed-point combinator:

```plaintext
Y := \lambda F. (\lambda x. F (x x)) (\lambda x. F (x x)).
```

329
With this, for any $\lambda$-expression $F$, the $\lambda$-application $Y F$ becomes the fix-point of $F$; therefore, $F (Y F) \equiv_\beta Y F$. Thus, we have a constructive way to build fixed-points for any $\lambda$-expression $F$. Note that $F$ does not need to be a $\lambda$-abstraction.

Let us now return to the recursive definition of factorial in $\lambda$-calculus enriched with integers. For this particular definition, let us define the $\lambda$-expression:

$$F := \lambda f . \lambda x . (\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x - 1))$$

The recursive definition of factorial is therefore a fixed-point of $F$, namely, $Y F$. It is such a fixed-point $\lambda$-expression that the letrec functional language construct in the definition of factorial refers to!

Let us experiment with this $\lambda$-calculus definition of factorial, by calculating factorial of 3:

$$\begin{align*}
(Y F) 3 & \equiv_\beta \\
F (Y F) 3 & = \\
(\lambda f . \lambda x . (\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x - 1))) (Y F) 3 & \Rightarrow_eta \\
\text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (Y F)(3 - 1) & \\
3 * (((Y F) 2) & \equiv_\beta \\
\ldots & \\
6 * ((Y F) 0) & \equiv_\beta \\
6 * (F (Y F) 0) & = \\
6 * (((\lambda f . \lambda x . (\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x - 1))) (Y F) 0) & \Rightarrow_\beta \\
6 * \text{if } 0 == 0 \text{ then } 1 \text{ else } 0 * (Y F)(0 - 1) & \\
6 * 1 & \\
6 & 
\end{align*}$$

Therefore, $\lambda$-calculus can be regarded as a simple programming language, providing support for functions, numbers, data-structures, and recursion. It can be shown that any computable function can be expressed in $\lambda$-calculus in such a way that its computation can be performed by $\beta$-reduction. This means that $\lambda$-calculus is a Turing-complete model of computation.

There are two aspects of $\lambda$-calculus that lead to complications when one wants to implement it.

One is, of course, the substitution: efficiency and correctness are two opposing tensions that one needs to address in any direct implementation of $\lambda$-calculus.

The other relates to the strategies of applying $\beta$-reductions: so far we used what is called full $\beta$-reduction, but other strategies include normal evaluation, call-by-name, call-by-value, etc. There are $\lambda$-expressions whose $\beta$-reduction does not terminate under one strategy but terminates under another. Moreover, depending upon the strategy of evaluation employed, other fixed-point combinators may be more appropriate.

Like $\beta$-reduction, the evaluation of expressions is confluent in many pure functional languages. However, once a language allows side effects, strategies of evaluation start playing a crucial role; to avoid any confusion, most programming languages hardwire a particular evaluation strategy, most frequently call-by-value.

We do not discuss strategies of evaluation here. Instead, we approach the other delicate operational aspect of $\lambda$-calculus, namely the substitution. In fact, we show that it can be completely eliminated if one applies a systematic transformation of $\lambda$-expressions into expressions over a reduced set of combinators.

More precisely, we show that any closed $\lambda$-expression can be systematically transformed into a $\lambda$-expression build over only the combinators $K := \lambda xy.x$ and $S := \lambda xyz.xz(yz)$, together with the $\lambda$-application operator. For example, the identity $\lambda$-abstraction $\lambda x.x$ is going to be $SKK$; indeed,

$$SKK \equiv_\beta \lambda z.Kz(Kz) = \lambda z.(\lambda xy.x)z(Kz) \equiv_\beta \lambda z.z \equiv_{\alpha} \lambda x.x.$$
Interestingly, once such a transformation is applied, one will not need the machinery of $\lambda$-calculus and $\beta$-reduction anymore. All we’ll need to do is to capture the contextual behavior of $K$ and $S$, which can be defined equationally very elegantly: $KXY = X$ and $SXYZ = XZ(YZ)$, for any other $KS$-expressions $X$, $Y$, $Z$.

Before we do that, we need to first discuss two other important aspects of $\lambda$-calculus: $\eta$-equivalence and extensionality.

### 4.5.8 Eta Equivalence and Extensionality

Here we discuss two common but equivalent extensions of $\lambda$-calculus, both concerned with additional proof support for equality of $\lambda$-expressions (besides $\alpha$-conversion and $\beta$-equivalence). The former is called $\eta$-equivalence and allows us to state that expressions $\lambda x. E\ x$ and $E$ are always equivalent, provided that $x$ does not occur free in $E$. The latter is called extensionality and allows us to derive that $E$ and $E'$ are equal provided that $E\ x$ and $E'\ x$ are equal for some variable $x$ that does not occur free in any of $E$ and $E'$.

#### Eta Equivalence

Let us consider the $\lambda$-expression $\lambda x. E\ x$, where $E$ is some $\lambda$-expression that does not contain $x$ free. Intuitively, $\lambda x. E\ x$ does nothing but wraps $E$: when called, it passes its argument to $E$ and then passes back $E$’s result. When applied on some $\lambda$-expression, say $E'$, note that $\lambda x. E\ x$ and $E$ behave the same. Indeed, since $E$ does not contain any free occurrence of $x$, one can show that $(\lambda x. E\ x)E' \equiv_\beta EE'$. Moreover, if $E$ is a $\lambda$-abstraction, say $\lambda y. F$, then $\lambda x. E\ x = \lambda x. (\lambda y. F)\ x \equiv_\beta \lambda x. F[x/y]$. The latter is $\alpha$-equivalent to $\lambda y. F$, so it follows that in this case $\lambda x. E\ x$ is $\beta$-equivalent to $E$.

Even though $\lambda x. E\ x$ and $E$ have similar behaviors in applicational contexts and they can even be shown $\beta$-equivalent when $E$ is a $\lambda$-abstraction, there is nothing to allow us to use their equality as an axiom in our equational inferences. In particular, there is no way to show that the combinator $\lambda x. \lambda y. xy$ is equivalent to $\lambda x.x$. To increase the proving capability of $\lambda$-calculus, still without jeopardizing its basic intuitions and applications, we consider its extension with the following equation:

$$(\eta) \quad \lambda x. E\ x = E,$$

for any $x \notin FV(E)$. We let $E \equiv_\beta \eta E'$ denote the fact that the equation $E = E'$ can be proved using standard equational deduction from all the equational axioms above: $(\alpha)$, $(\beta)$, $(\eta)$, plus those for substitution. The relation $\equiv_\beta \eta$ is also called $\beta\eta$-equivalence. The $\lambda$-calculus enriched with the rule $(\eta)$ is also written $\lambda + \eta$.

#### Extensionality

Extensionality is a deduction rule encountered in several branches of mathematics and computer science. It intuitively says that in order to prove two objects equal, one may first extend them in some rigorous way. The effectiveness of extensionality comes from the fact that it may often be the case that the extended versions of the two objects are easier to prove equivalent.

Extensionality was probably first considered as a proof principle in set theory. In naive set theory, sets are built in a similar fashion to Peano numbers, that is, using some simple constructors (together with several constraints), such as the empty set $\emptyset$ and the list constructor $\{x_1, \ldots, x_n\}$. Thus, $\{\emptyset, \emptyset, \{\emptyset\}\}$ is a well-formed set. With this way of constructing sets, there may be the case that two sets with the same elements have totally different representations. Consequently, it is almost impossible to prove any meaningful property on sets, such as distributivity of union and intersection, etc., by just taking into account how sets are constructed. In particular, proofs by structural induction are close to useless.
Extensionality is often listed as the first axiom in any axiomatization of set theory. In that context, it basically says that two sets are equal iff they have the same elements. Formally, sets $S$ and $S'$ are equal whenever we can prove the statement “for any element $x$, $x \in S$ iff $x \in S'$”. Therefore, in order to show sets $S$ and $S'$ equal, we can first extend them (regarded as syntactic terms) by applying them the membership operator. In most cases the new task is easier to prove.

Similarly, the principle of extensionality applies to functions: two functions are equal whenever they evaluate to the same values for the same elements. Formally, functions $f$ and $g$ are equal whenever we can prove the statement “for any element $x$, $f(x) = g(x)$”. We again have to prove a syntactically extended task, but like for sets it is often the case that the extended task is simpler to prove.

In $\lambda$-calculus, extensionality takes the following shape:

\[(\text{ext}) \quad \text{If } E x = E' x \text{ for some } x \notin \text{FV}(EE'), \text{ then } E = E'.\]

Therefore, two $\lambda$-abstractions are equal if they are equal when applied on some variable that does not occur free in any of them. Note that “for some $x'$” can be replaced by “for any $x$" in ext. We let $E \equiv_{\text{ext}} E'$ denote the fact that the equation $E = E'$ can be proved using standard equational deduction using ($\alpha$) and ($\beta$), together with ext. The $\lambda$-calculus extended with ext is also called $\lambda + \text{ext}$.

The following important result says the extensions of $\lambda$-calculus with ($\eta$) and with ext are equivalent:

**Theorem 25.** $\lambda + \eta$ is equivalent to $\lambda + \text{ext}$.

**Proof.** In order to show that two mathematical theories are equivalent, one needs to show two things: (1) how the syntax of one translates into the syntax of the other, or in other words to show how one can mechanically translate assertions in one into assertions in the other, and (2) that all the axioms of each of the two theories can be proved from the axioms of the other, along the corresponding translation of syntax. In our particular case of $\lambda + \eta$ and $\lambda + \text{ext}$, syntax remains unchanged when moving from one logic to another, so (1) above is straightforward. We will shortly see another equivalence of logics, where (1) is rather involved. Regarding (2), all we need to show is that under the usual $\lambda$-calculus with ($\alpha$) and ($\beta$), the equation ($\eta$) and the principle of extensionality are equivalent.

Let us first show that ($\eta$) implies ext. For that, let us assume that $E x \equiv_{\beta\eta} E' x$ for some $\lambda$-expressions $E$ and $E'$ and for some variable $x \notin \text{FV}(EE')$. We need to show that $E \equiv_{\beta\eta} E'$:

$$E \equiv_{\beta\eta} \lambda x.E x \equiv_{\beta\eta} \lambda x.E' x \equiv_{\beta\eta} E'.$$

Note the use of $\equiv_{\beta\eta}$ in the equivalences above, rather than just $\equiv_{\beta}$. That is because, in order to prove the axioms of the target theory, $\lambda + \text{ext}$ in our case, one can use the entire calculus machinery available available in the source theory, $\lambda + \eta$ in our case.

Let us now prove the other implication, namely that ext implies ($\eta$). We need to prove that $\lambda x.E x \equiv_{\text{ext}} E$ for any $\lambda$-expression $E$ and any $x \notin \text{FV}(E)$. By extensionality, it suffices to show that $(\lambda x.E x)x \equiv_{\text{ext}} E x$, which follows immediately by $\beta$-equivalence because $x$ is not free in $E$. \hfill \Box

### 4.5.9 Combinatory Logic

Even though $\lambda$-calculus can be defined equationally and is a relatively intuitive framework, as we have noticed several times by now, substitution makes it non-trivial to implement effectively. There are several approaches in the literature addressing the subtle problem of automating substitution to avoid variable capture, all with their advantages and disadvantages. We here take a different approach. We show how $\lambda$-expressions can be
automatically translated into expressions over combinators, in such a way that substitution will not even be needed anymore.

A question addressed by many researchers several decades ago, still interesting today and investigated by many, is whether there is any simple equational theory that is entirely equivalent to $\lambda$-calculus. Since $\lambda$-calculus is Turing complete, such a simple theory may provide a strong foundation for computing.

Combinatory logic was invented by Moses Schönfinkel in 1920. The work was published in 1924 in a paper entitled “On the building blocks of mathematical logic”. Combinatory logic is a simple equational theory over two sorts, $\text{Var}$ and $\text{Exp}$ with $\text{Var} < \text{Exp}$, a potentially infinite set $x, y, \text{etc.}$, of constants of sort $\text{Var}$ written using lower-case letters, two constants $K$ and $S$ of sort $\text{Exp}$, one application operation with the same syntax and left-associativity parsing convention as in $\lambda$-calculus, together with the two equations

$$KXY = X,$$
$$SXYZ = XZ(YZ),$$

quantified universally over $X, Y, Z$ of sort $\text{Exp}$. The constants $K$ and $S$ are defined equationally in such a way to capture the intuition that they denote the combinators $\lambda xy.x$ and $\lambda xyz.xz(yz)$, respectively. The terms of the language, each of which denoting a function, are formed from variables and constants $K$ and $S$ by a single construction, function application. For example, $S(SxKS)yz(SKxK)z$ is a well-formed term in combinatory logic, denoting some function of free variables $x, y, \text{and } z$.

Let $\text{CL}$ be the equational theory of combinatory logic above. Note that a function $FV$ returning the free variables that occur in a term in combinatory logic can be defined in a trivial manner, because there are no bound variables in $\text{CL}$. Also, note that the extensionality principle from $\lambda$-calculus translates unchanged to $\text{CL}$:

(ext) If $Ex = E'x$ for some $x \notin FV(EE')$, then $E = E'$.

Let $\text{CL} + \text{ext}$ be $\text{CL}$ enriched with the principle of extensionality. The following is a landmark result:

**Theorem 26.** $\lambda + \text{ext}$ is equivalent to $\text{CL} + \text{ext}$.

**Proof.** Let us recall what one needs to show in order for two mathematical theories to be equivalent: (1) how the syntax of one translates into the syntax of the other; and (2) that all the axioms of each of the two theories can be proved from the axioms of the other, along the corresponding translation of syntax.

Let us consider first the easy part: $\lambda + \text{ext}$ implies $\text{CL} + \text{ext}$. We first need to show how the syntax of $\text{CL} + \text{ext}$ translates into that of $\lambda + \text{ext}$. This is easy and it was already mentioned before: let $K$ be the combinator $\lambda xy.x$ and let $S$ be the combinator $\lambda xyz.xz(yz)$. We then need to show that the two equational axioms of $\text{CL} + \text{ext}$ hold under this translation: they can be immediately proved by $\beta$-equivalence. We also need to show that the extensionality in $\text{CL} + \text{ext}$ holds under the above translation: this is obvious, because it is exactly the same as the extensionality in $\lambda + \text{ext}$.

Let us now consider the other, more difficult, implication. So we start with $\text{CL} + \text{ext}$, where $K$ and $S$ have no particular meaning in $\lambda$-calculus, and we need to define some map that takes any $\lambda$-expression and translates it into an expression in $\text{CL}$.

To perform such a transformation, let us add syntax for $\lambda$-abstractions to $\text{CL}$, but without any of the equations of $\lambda$-calculus. This way one can write and parse $\lambda$-expressions, but still have no meaning for those. The following ingenious *bracket abstraction* rewriting system transforms any uninterpreted $\lambda$-expression into an expression using only $K, S$, and the free variables of the original $\lambda$-expression:
1. $\lambda x.\rho \Rightarrow [x]\rho$

2. $[x]y \Rightarrow \begin{cases} SKK & \text{if } x = y \\ Ky & \text{if } x \neq y \end{cases}$

3. $[x](\rho\rho') \Rightarrow S([x]\rho)([x]\rho')$

4. $[x]K \Rightarrow KK$

5. $[x]S \Rightarrow KS$

The first rule removes all the $\lambda$-bindings, replacing them by corresponding bracket expressions. Here $\rho$ and $\rho'$ can be any expressions over $K$, $S$, variables, and the application operator, but also over the $\lambda$-abstraction operator $\lambda_- : \text{Var} \to \text{Exp}$. However, note that rules 2-5 systematically eliminate all the brackets. Therefore, the bracket abstraction rules above eventually transform any $\lambda$-expression into an expression over only $K$, $S$, variables, and the application operator.

The correctness of the translation of $\lambda + \text{ext}$ into $\text{CL} + \text{ext}$ via the bracket abstraction technique is rather technical: one needs to show that the translated versions of equations in $\lambda$ can be proved (by structural induction) using the machinery of $\text{CL} + \text{ext}$.

Exercise 185. (Technical) Prove the correctness of the translation of $\lambda + \text{ext}$ into $\text{CL} + \text{ext}$ above.

We do not need to understand the details of the proof of correctness in the exercise above in order to have a good intuition on why the bracket abstraction translation works. To see that, just think of the bracket abstraction as a means to associate equivalent $\lambda$-expressions to other $\lambda$-abstractions, within the framework of $\lambda$-calculus, where $K$ and $S$ are their corresponding $\lambda$-expressions. As seen above, it eventually reduces any $\lambda$-expression to one over only combinators and variables, containing no explicit $\lambda$-abstractions except those that define the combinators $K$ and $S$. To see that the bracket abstraction is correct, we can think of each bracket term $[x]E$ as the $\lambda$-expression that it was generated from, $\lambda x. E$, and then show that each rule in the bracket abstraction transformation is sound within $\lambda$-calculus. For example, rule 3 can be shown by extensionality: $(\lambda x.\rho\rho')z \equiv_\beta (\rho(z/x)(\rho'(z/x))) \equiv_\beta ((\lambda x.\rho)z)((\lambda x.\rho')z) \equiv_\beta (\lambda xyz.z(xz))(\lambda x.\rho)(\lambda x.\rho')z = S(\lambda x.\rho)(\lambda x.\rho')z$, so by extensionality, $\lambda x.\rho\rho' \equiv_\text{ext} S(\lambda x.\rho)(\lambda x.\rho')$.

This way, one can prove the soundness of each of the rules in the bracket abstraction translation. As one may expect, the tricky part is to show the completeness of the translation, that is, that everything one can do with $\lambda$-calculus and $\text{ext}$ can also do with with its subcalculus $\text{CL} + \text{ext}$. This is not hard, but rather technical.

Exercise 186. Define the bracket abstraction translation above formally in Maude. To do it, first define $\text{CL}$, then add syntax for $\lambda$-abstraction and bracket to $\text{CL}$, and then add the bracket abstraction rules as equations (which are interpreted as rewrite rules by Maude). Convince yourself that substitution is not a problem in $\text{CL}$, by giving an example of a $\lambda$-expression which would not be reducible with the definition of $\lambda$-calculus in Exercise ??, but whose translation in $\text{CL}$ can be reduced with the two equations in $\text{CL}$.

---

some additional notes:

In the pure system there is a small set of constants all of which are themselves pure combinators, usually the primitive combinators are named $S$, $K$ and $I$, though $I$ is in fact definable in terms of $S$ and $K$.

334
The intended meaning of these constants can be described simply by showing a corresponding transformation or reduction which the constant may be thought of as effecting.

The transformations are as follows: $I\ x = x$ $K\ x\ y = x$ $S\ f\ g\ x = (f\ x)(g\ x)$ This set of combinators has the property of combinatory completeness which means that for any function definition of the form: $f\ x =$ combinatorial expression involving $x$

There exists an expression $E$ in $S,K$ and $I$ such that: $f = E$

This property establishes the close relationship between Pure Combinatory Logic and the Pure lambda Calculus, in which a special notation for functional abstraction is available. It shows that the notation for functional abstraction, though a great convenience, adds no expressiveness to the language.

As well as having combinatorial completeness, Pure Combinatory Logic is able to express all effectively computable functions over natural numbers appropriately represented as combinators.


This is the first published work in the field which came later to be known as combinatoric logic.

Schnfinkel shows how the use of bound variables in logic can be dispensed with. The use of higher order functions makes possible the reduction of logic to a language consisting of one constructor (the application of a function to an argument) and three primitive constants $U$, $C$ (now usually called $K$) and $S$. A function is termed “higher order” if it will accept a function as an argument, or return one as its result. $U$, $C$, and $S$ are all higher order functions.

The meaning of these primitives may be understood through the following informal definitions:

$C\ xy = (C(x))(y) = x$ $C$ is a function which given any value $x$, returns the constant $x$ valued function.

$S\ fgx = ((S(f))(g))(x) = (f(x))(g(x))$ $S$ is a function which combines two functions, say $f$ and $g$, supplied as successive arguments. The resulting function, given a value $x$, returns the value obtained by applying the value of $f$ at $x$ to the value of $g$ at $x$. $U\ PQ = (U\ P)(Q) = \forall\ x\ .\ \neg\ (P\ x\ and\ Q\ x\ )$ $U$ is a generalised scheffer stroke. It should be thought of as applying to two predicates and returns the universal quantification of the negated conjunction of the two predicates.

These combinators are sufficient to enable arbitrary first order predicates to be expressed without the use of bound variables, which appear in first order logic whenever a quantifier is used. This can be demonstrated most straightforwardly using a simple algorithm which converts lambda-expressions to combinators. They are not limited to first-order predicates, but without some constraints (equivalent to those found in first order logic, or more liberally and appropriately to those in Church’s Simple Theory of Types [Church40]) the logic which results from the use of these combinators is at risk of proving inconsistent. Combinatory logic has traditionally tried to avoid type constraints and has therefore been dogged with difficulties in achieving strength and consistency.

Schnfinkel’s paper remains an accessible introduction to combinatory logic which makes clear the original motivation for this innovation. This original motivation was vigorously pursued later by H.B.Curry and his collaborators. The results of Curry’s programme were published in Combinatory Logic, Vols 1 & 2.

While Schfinkel was successful in showing that combinators could provide plausible notation, the semantic and proof theoretic aspects were more difficult to put in place. The difficulty is more pronounced if it hoped to use combinatory logic as a foundation for mathematics, than it is for the use of combinators in less demanding roles (e.g. for programming languages).

The Pure combinatory logic may be thought of as Schnfinkels system with the logical operator “$U$” omitted (though the presentation may vary). Illative combinatory logics are formed by adding additional logical primitives to the pure combinators. There are many illative combinatory logics, which vary among themselves in substance as well as in presentation.
Pure combinatory logic is so closely related to Church’s lambda-calculus that it is best studied alongside the lambda-calculus, for which the most comprehensive modern text is probably The Lambda Calculus. A very popular and entertaining introduction to the pure combinators may be found in To Mock a Mockingbird.

Research on illative combinatory logics has been continued by Curry’s students, Roger Hindley and Jonathan Seldin, and by Martin Bunder. Hindley, Lercher and Seldin published An Introduction to Combinatory Logic, an excellent and readable short introduction to the subject, now superseded by the more comprehensive An Introduction to Combinators and the Lambda-Calculus.

Recently Randall Holmes has developed logical systems based on combinatory logic and the lambda calculus using methods derived from Quine’s formulations of set theory.

The attempt to make a foundation for mathematics using the illative combinatory logic were blighted by Curry’s paradox and semantic opacity. While systems avoiding this paradox have been devised, they are typically weak, and lack the kind of convincing semantic intuition found in first order set theory.


4.5.10  De Bruijn Nameless Representation

Interestingly, note that the morphism obtained above contains no references to the variables that occur in the original \( \lambda \)-expression. It can be shown that the interpretation of \( \lambda \)-expressions into a CCC is invariant to \( \alpha \)-conversion. To see that, let us draw the morphism above as a tree, where we write \( \lambda \) instead of curry, \( \langle , \rangle \) instead of \( \langle , \rangle \); \( \text{app} \langle \rangle \langle \rangle \), \( \langle , , \rangle \) instead of the remaining \( \langle , , \rangle \) and \( i \) instead of \( \pi_i \) (and omit the types):

![Diagram](image)

The (right) tree above suggests a representation of \( \lambda \)-expressions that is invariant to \( \alpha \)-conversion: each binding variable is replaced by a natural number, representing the number of \( \lambda \)-s occurring on the path to it; that number then replaces consistently all the bound occurrences of the variable. The corresponding lambda expression without variables obtained using this transformation is \( \lambda (\lambda \lambda (3 (1, (2 1))) \lambda) 2 \).

**Exercise 187.** Explain why this representation is invariant to \( \alpha \)-conversion.

The representation of \( \alpha \)-expressions above was explicitly proposed as a means to implement \( \lambda \)-calculus in 1971 by Nicholas de Bruijn.

In the same paper, de Bruijn proposed another encoding which became more popular. We do not know whether de Bruijn was influenced by the CCC interpretation of \( \lambda \)-expressions or not, but we discuss his other representation technique here.

**de Bruijn Nameless Representation of \( \lambda \)-expression**
The second and more popular representation technique of $\lambda$-expressions proposed by Nicholas de Bruijn in 1971 is a bottom-up version of the above representation. For the above example the tree representing the encoding is (we omit the types):

$$\lambda x \lambda f \lambda y \lambda g y g (\ldots) x f x$$

In this encoding, each variable is replaced by the number of lambda abstractions on the path from it to the lambda abstraction binding it. The encoding for the given example is $\lambda (\lambda (0 (2, (1 2))) \lambda 0)$.

One can easily define application for the above de Bruijn encoding:

$$(\lambda E E') \Rightarrow E[E'/0]$$

$$(E E')[E''/i] \Rightarrow (E[E''/i]) (E'[E''/i])$$

$$(\lambda E)[E'/i] \Rightarrow \lambda (E[E'/i (1 + i_0)])$$

$$(E E'/i) \Rightarrow \text{if } j = i \text{ then } E \text{ else (if } j > i \text{ then } j - 1 \text{ else } j \text{ fi) } fi$$

$$(\uparrow E') \Rightarrow \uparrow^n E$$

$$(\uparrow (E E')) \Rightarrow (\uparrow E') (\uparrow^i E)$$

$$(\uparrow (\lambda E) \Rightarrow \lambda (\uparrow^1 E)$$

$$(\uparrow j \Rightarrow \text{if } j > i \text{ then } (j + 1) \text{ else } j \text{ fi})$$

Exercises

Exercise 188. Define the transformation above formally in Maude.

Exercise 189. Define application for the first de Bruijn encoding, in a similar style to the one above.

4.5.11 Notes

4.5.12 Exercises

Exercise 190. ★ Define the syntax of $\lambda$-calculus in a Maude module using mix-fix notation. Then parse the various $\lambda$-expressions that were discussed in this section.

Exercise 191. ★ Extend the Maude definition of $\lambda$-calculus in Exercise [190] with a definition of free variables. You should define an operation $\mathcal{FV}$ taking an expression and returning a set of variables.

Exercise 192. Prove the following equivalences of $\lambda$-expressions:

- $\lambda x.x \equiv_\alpha \lambda y.y$,
- $\lambda x.(\lambda y.y) \equiv_\alpha \lambda y.(\lambda x.x)$,
- $\lambda x.(\lambda y.y) \equiv_\alpha \lambda y.(\lambda y.y)$.

Exercise 193. Show that $(\lambda x.(\lambda y.x))yx \equiv_\beta y$. 337
Exercise 194. ★ Define substitution in Maude, as a partial operation. As discussed in Section 4.5.3, a partial substitution applies only if it does not lead to a variable capture; otherwise the substitution operation is undefined. In other words, when applying a partial substitution we do not need to perform any α-conversions.

Hint. Define \((\lambda y. E)[E'/x]\) when \(y \neq x\) as a conditional equation, with condition \(y \notin FV(E')\).

Exercise 195. ★ Define unrestricted substitution in Maude, which takes into account and avoids variable capture. Use the core-level Maude capabilities for this exercise (see also Exercise 196).

Hint. You need a mechanism to generate fresh variables. For example, you may define an operation \(\text{var} : \text{Int} \rightarrow \text{Var}\), where \(\text{var}(1), \text{var}(2), \text{etc.}\), represent fresh variables. A simple but wasteful solution when calculating \((\lambda y. E)[E'/x]\) is to always replace \(y\) in \(\lambda y. E\) by a fresh variable before applying the substitution; a better solution is to only do it when \(y \in FV(E')\).

Exercise * 196. ★ Define unrestricted substitution in Maude, using the meta-level capabilities of Maude (see also Exercise 195). You should traverse the meta-term and propagate the substitution operation through any operation label which is not a \(\lambda\)-binding. For \(\lambda\)-bindings you should apply the same conceptual operation as in Exercise 195. The advantage of the meta-level approach is that it is more modular that the other approaches. We will be able to add new constructs to \(\lambda\)-calculus, for example builtins, without having to change the definition of the substitution.

Exercise 197. ★ Define \(\lambda\)-calculus formally in Maude, using each of the substitution approaches in Exercises 194, 195, and 196. For the former approach (partial substitution), show that there are \(\lambda\)-expressions that cannot be \(\beta\)-reduced automatically with your definition of \(\lambda\)-calculus to a normal form, even though they are closed (or combinators) and all the binding variables are initially distinct from each other.

Exercise 198. Define the other Boolean operations which were not already defined in Section 4.5.6 (including at least \(\text{or}\), \(\text{not}\), \(\text{implies}\), \(\text{iff}\), and \(\text{xor}\)) as \(\lambda\)-expressions.

Exercise 199. Prove that the Church Boolean operators defined in Exercise 198 have all their desired properties (a set of such desired properties can be found in the Maude BOOL module; use the command “show module BOOL” lists them).

Exercise 200. With the notions and notations in Section 4.5.6, show that for any Church numeral \(n_1\), both \(\text{succ} n_1\) and \(\text{succ}' n_1\) represent the same numeral, namely \((n + 1)_1\).

Hint. Induction on the structure of \(n_1\).

Exercise 201. Prove that Church numerals form indeed a model of natural numbers, by showing the two properties derived from Peano’s axioms (see Section 4.5.6).

Exercise 202. Define multiplication on Church numerals and prove its Peano properties.

Hint. Multiplication can be defined several different interesting ways.

Exercise 203. Define the power operator (raising a number to the power of another) using Peano-like axioms. Then define power on Church numerals and show that it satisfies its Peano axioms.

Exercise 204. Add builtin integers to \(\lambda\)-calculus in Maude, extending Exercise 197 based on substitution as defined in Exercise 196.

Hint. The only change needed should be to subsort Int to Exp. Since Maude’s parsing is at the level of kinds, instead of sorts, we should not need to overload the Int operations to take and return Exp sorts.
4.6 Simply-Typed Lambda Calculus

We now discuss a non-trivial extension of $\lambda$-calculus with types. The idea is that each variable binding is assigned a type, which will allow one to calculate a unique type for each well-formed $\lambda$-expression or $\lambda$-term.

As we know from our experience with programming languages, the addition of types will allow one to reject programs that are not well-typed, with the intuition that those programs are most likely wrong (with respect to what the programmer meant).

Typing comes at a price: sometimes correct programs are rejected. One can, of course, argue that those programs are not correct (by definition, because they do not type). All in all, practice has shown that typing is overall useful in programming languages. Simply typed $\lambda$-calculus is perhaps the simplest typed language.

4.6.1 Syntax

The BNF syntax of simply-typed $\lambda$-calculus is

$$
\text{Var} ::= x \mid y \mid \ldots \\
\text{Type} ::= \bullet \mid \text{Type} \to \text{Type} \\
\text{Exp} ::= \text{Var} \mid \lambda \text{Var} : \text{Type} . \text{Exp} \mid \text{ExpExp}.
$$

To keep the presentation simple, for the time being we assume only one constant type, $\bullet$, and only one type constructor, $\to$. Thus $(\bullet \to \bullet) \to (\bullet \to (\bullet \to \bullet))$ is a well-formed type. To simplify writing, we assume that $\to$ is right-associative; the type above can then be written $(\bullet \to \bullet) \to \bullet \to \bullet \to \bullet$. As in the case of untyped $\lambda$-calculus, the $\lambda$-application is still assumed to be left-associative.

Exercise 205. Define the syntax above in a Maude module, using the alternative mix-fix notation.

4.6.2 Terms and Type Checking

Using the syntax above, one can naturally generate simply-typed terms or program, such as, for example, $\lambda x : \bullet . \lambda f : \bullet \to \bullet . f x$. The intuition for this $\lambda$-abstraction is that it takes some $\lambda$-expressions of types $\bullet$ and $\bullet \to \bullet$, respectively, and applies the latter on the former. Naturally, the type of the result is expected to be $\bullet$.

Unlike in the case of untyped $\lambda$-calculus, the BNF (or, equivalently, the mix-fix notation) is not powerful enough to express all the intended well-formed, or better say well-typed, terms. Indeed, for example the term $\lambda x : \bullet . x x$ does not make any sense, because $x$, in order to be applied on an expression of type $\bullet$, in particular on itself, must have the type $\bullet \to s$ for some type $s$; however, $x$ is declared of type $\bullet$. Moreover, one can formally show that $\lambda x : s . x x$ is not well-formed for any type $s$.

Even more, it can be shown that there is no context free grammar (CFG) whose language consists of all well-typed $\lambda$-expressions. This is perhaps the simplest language supporting the folklore claim that “well-formed programs do not form a context-free language”.

Now the natural question is how to characterize the well-formed simply-typed $\lambda$-expressions. There are three equivalent approaches to do this, all being easily adaptable to other typed frameworks. Let us first introduce some important notation.

A type assignment is a finite set $X = \{x_1 : s_1, \ldots, x_n : s_n\}$ of pairs $x : s$, where $x$ is a variable and $s$ a type, with the property that each variable occurs at most once in $X$: it is not possible to have $x : s_1, x : s_2 \in X$.
different types $s_1$, $s_2$. One of the main reasons for this limitation is that well-formed $\lambda$-expressions, including those which are just variables, are desired to have unique types. Then if $x$ occurs in $X$, written (admittedly ambiguously) $x \in X$, we may let $X(x)$ denote the type $s$ such that $x : s \in X$. Often the curly brackets "{" and "}" are omitted from the notation of $X$ and its elements are permuted conveniently; thus, if $x \notin X$, then $X, x : s$ is a type assignment containing $x : s$. We let $\text{TypeAssignment}$ denote the set of type assignments.

For the time being, let us introduce the notation $X \Rightarrow E : t$, typically called a type judgment, or a well-typed $\lambda$-expression or $\lambda$-term (sometimes “well-typed” may be dropped if understood), with the intuition that under the type assignment $X$, the $\lambda$-expression $E$ is well-typed and has the type $t$. For example, one can write $x : \cdot, f : \cdot \rightarrow \cdot \Rightarrow fx : \cdot$. We will shortly see three different ways to define this intuition precisely.

Alternative notations for $X \Rightarrow E : t$ could be $(\forall X)E : t$ or $(E : t)X$, or just $E : t$ when $X$ is understood from context, or even simply $E$ if both $X$ and $t$ are understood. Let us next discuss the three different (but related) formal approaches to define this.

1. Proof system

We can define a proof system that can derive precisely the well-typed $\lambda$-expressions. The following three rules do this:

$$
\begin{align*}
X, x : s & \Rightarrow x : s & \text{for any type assignment } X, x : s; \\
X, x : s & \Rightarrow E : t & \text{for any type assignment } X, x : s, \text{ any Exp-term } E, \text{ and any type } t; \\
X & \Rightarrow E : s \rightarrow t & X \Rightarrow E' : s & \text{for any type assignment } X, \text{ Exp-terms } E, E', \text{ and types } s, t.
\end{align*}
$$

As usual, $X \Rightarrow E : t$ is called derivable if there is some sequence $X_1 \Rightarrow E_1 : t_1, \ldots, X_n \Rightarrow E_n : t_n$ such that $X_n \Rightarrow E_n : t_n$ is $X \Rightarrow E : t$ and each $X_i \Rightarrow E_i : t_i$ follow by one of the three rules above from previously ($< i$) derived well-typed terms. We may write $\Rightarrow X \Rightarrow E : t$, or sometimes even just $X \Rightarrow E : t$ when $X$ is understood from context.

**Exercise 206.** Derive $x : \cdot \Rightarrow \lambda f : \cdot \rightarrow \cdot.fx : \cdot$. Also, derive $\emptyset \Rightarrow \lambda x : \cdot.\lambda f : \cdot \rightarrow \cdot \rightarrow \cdot \rightarrow fxx : \cdot \rightarrow \cdot$

When $X$ is empty, we write $E : t$ instead of $\emptyset \Rightarrow E : t$. We are not going to (re)define the notions of free variable, with the corresponding operator $\text{FV}$, and substitution. They have precisely the same meaning as in untyped $\lambda$-calculus.

Properties about well-typed $\lambda$-expressions are typically proved by induction on the length of derivation.

**Proposition 14.** The following hold:

- If $X \Rightarrow E : t$ then $\text{FV}(E) \subseteq X$;
- If $X \Rightarrow \lambda x : s. E : s \rightarrow t$ then $X, y : s \Rightarrow (E[y/x]) : t$ for any $y \notin X$;
- If $X \Rightarrow E : s$ and $X \Rightarrow E : t$ then $s = t$;
- If $X, x : s \Rightarrow E : t$ and $X \Rightarrow E' : s$ then $X \Rightarrow E'[x/x] : t$;
- If $X, X'$ are type assignments and $E$ is a $\lambda$-expression such that for all $x \in \text{FV}(E)$, $x : s \in X$ iff $x : s \in X'$, then $X \Rightarrow E : t$ iff $X \Rightarrow E' : t$;
- $\lambda x : s.xx$ does not type.
2. Typing Algorithm

We can also define a relatively trivial typing algorithm that takes a type assignment \( X \) together with a \( \lambda \)-expression \( E \), and tries to calculate a type \( t \) for \( E \). The algorithm traverses \( E \) recursively:

Algorithm \( \mathcal{A}(X, E) \)

- if \( E \) is \( x \) and \( x : s \in X \) then return \( s \);
- if \( E \) is \( \lambda x : s. E' \) and \( x \notin X \) and \( \mathcal{A}((X, x : s), E') \) returns \( t \) then return \( t \rightarrow s \);
- if \( E \) is \( E_1 E_2 \) and \( \mathcal{A}(X, E_1) \) returns \( s \rightarrow t \) and \( \mathcal{A}(X, E_2) \) returns \( s \) then return \( t \);
- otherwise return error

Exercise 207. Prove that \( X \mathrel{\bowtie} E : t \) is derivable if and only if \( \mathcal{A}(X, E) \) returns \( t \). (Hint: By structural induction on \( E \).)

3. Sets of Terms

Let us next give another characterization of the well-typed \( \lambda \)-terms. We define the family of sets \( \{ T_t(X) \}_{X \in \text{TypeAssignment}, t \in \text{Type}} \) as the (componentwise) smallest set \( \{ W_{X,t} \}_{X \in \text{TypeAssignment}, t \in \text{Type}} \) of words in the (CF) language of \( \lambda \)-calculus such that:

- \( x \in W_{X,s} \) if \( x : s \in X \)
- \( \lambda x : s. E \in W_{X,s \rightarrow t} \) if \( E \in W_{(X,x:s),t} \)
- \( E_1 E_2 \in W_{X,t} \) if \( E_1 \in W_{X,s \rightarrow t} \) and \( E_2 \in W_{X,s} \) for some \( s \in \text{Type} \).

Exercise 208. Prove that \( X \mathrel{\bowtie} E : t \) is derivable iff \( E \in T_t(X) \).
(Hint: By structural induction on \( E \).)

4.6.3 Equational Rules

We have discussed so far techniques to check that \( \lambda \)-expressions are well-typed. From now on we assume that all the \( \lambda \)-expressions that occur in any context are well-typed. More precisely, whenever we write \( X \mathrel{\bowtie} E : t \), we assume that \( E \) is well-typed under the type assignment \( X \) and that it has the type \( t \). We now focus on equational properties of simply-typed \( \lambda \)-calculus. These equations play a dual role: on the one hand they give means to show programs equivalent, while on the other hand underlay the infrastructure necessary to define a canonical model of \( \lambda \)-calculus.

An equation is a 4-tuple consisting of a type assignment \( X \), two \( \lambda \)-expressions \( E \) and \( E' \), and a type \( t \), such that \( X \mathrel{\bowtie} E : t \) and \( X \mathrel{\bowtie} E' : t \). To simplify notation, we write such equations as \( (\forall X) E =_t E' \), with the intuition that for any interpretation of the variables in \( X \) (i.e., any assignment of values of corresponding type to variables in \( X \)), the expressions \( E \) and \( E' \) evaluate to the same value, which has the expected type \( t \).

A set of equations \( E \) is also called an equational theory (in \( \lambda \)-calculus). Given an equational theory \( E \) and an equation \( e \), we call the syntactic construct \( E \vdash e \) an equational judgment. We next give a set of derivation rules for equational judgments:
(axiom) \( E \vdash (\forall X) E =_t E' \) if \((\forall X) E =_t E' \) is in \( E \)

(add) \[
\frac{E \vdash (\forall X) E =_t E'}{E \vdash (\forall X, x : s) E =_t E'} \quad \text{if} \ x \notin X
\]

(reflexivity) \( E \vdash (\forall X) E =_t E \) if \( X \gg E : t \)

(symmetry) \[
\frac{E \vdash (\forall X) E =_t E'}{E \vdash (\forall X) E' =_t E}
\]

(transitivity) \[
\frac{E \vdash (\forall X) E =_t E' \quad E \vdash (\forall X) E' =_t E''}{E \vdash (\forall X) E =_t E''}
\]

(application) \[
\frac{E \vdash (\forall X) E_1 =_{s \rightarrow t} E_1' \quad E \vdash (\forall X) E_2 =_s E_2'}{E \vdash (\forall X) E_1 E_2 =_t E_1' E_2'}
\]

(\( \xi \)) \[
E \vdash (\forall X, x : s) E =_t E' \quad \frac{E \vdash (\forall X) \lambda x : s.E =_{s \rightarrow t} \lambda x : s.E'}{E \vdash (\forall X) \lambda x : s.E'}
\]

(\( \beta \)) \[
E \vdash (\forall X) (\lambda x : s.E)E' =_t E'[x/x] \quad \text{if} \ X, x : s \gg E : t \quad \text{and} \quad X \gg E' : s
\]

(\( \eta \)) \[
E \vdash (\forall X) \lambda x : s. E =_{s \rightarrow t} E \quad \text{if} \ x \notin FV(E)
\]

The rule (axiom) says that any equation already existing in \( E \) is derivable from \( E \). The rule (add) allows one to add typed variables to the type assignment of a derived equation; this is necessary for several technical reasons, such as, for example, to bring the two equations to the same type assignment in order to apply the (transitivity) or the (application) rule. The next four rules, (reflexivity), (symmetry), (transitivity), and (application), are self explanatory and are instances of general equational reasoning in algebraic specifications to the signature of \( \lambda \)-calculus; in particular, (application) is an instance of the congruence deduction rule. The rule (\( \xi \)) is almost an instance of the equational congruence rule to the \( \lambda \)-abstraction construct; note, however, that the type assignment needs to be changed appropriately. The last two rules, (\( \beta \)) and (\( \eta \)) are nothing but the equational rule and typed versions of the (\( \beta \)) and (\( \eta \)) equations from untyped \( \lambda \)-calculus.

**Proposition 15.** If \( E \vdash (\forall X) E =_t E' \) then \( X \gg E : t \) and \( X \gg E' : t \).

**Proposition 16.** If \( E, E' \) are two \( \lambda \)-expressions and \( x \notin X \) then 
\( E \vdash (\forall X) E =_t E' \) iif \( E \vdash (\forall X, x : s) E =_t E' \).

**Corollary 8.** If \( E, E' \) are \( \lambda \)-expressions and \( X, X' \) are type assignments such that \( x : s \in X \) iff \( x : s \in X' \) for any \( x \in FV(EE') \), then 
\( E \vdash (\forall X) E =_t E' \) iif \( E \vdash (\forall X') E =_t E' \).

**Proposition 17.** If \( E \vdash (\forall X) E =_s E' \) and \( Y, y : s \gg F : t \) such that \( X \cup Y \) is a proper type assignment, then 
\( E \vdash (\forall X \cup Y) F[E/y] =_t F[E'/y] \).

(Proof hint: Eliminate the substitution by applying the rule (\( \beta \)) twice backwards.)

Comment. In the next iteration of the class, allow well-typed terms of the form \( \lambda x : s. \lambda x : t.E \) and define \( \lambda \)-calculus as an instance of equational reasoning (including substitution) enriched with just (\( \beta \)) and (\( \eta \)).
4.6.4 Models

1. A Type-indexed set $M = \{M_t\}_{t \in \text{Type}}$ is an infinite collection of sets, one for each type; there is no relationship required among the sets $M_s$, $M_t$, and $M_{s \to t}$ for any types $s$ and $t$. Note that type assignments can be regarded as Type-indexed sets with only a finite number of sets non-empty. For example, the type assignment \{ $x : s, y : s, z : t$ \} can be regarded as the Type-indexed set whose $s$-component is \{ $x, y$ \}, whose $t$-component is \{ $z$ \}, and whose other components are all empty.

   For a given $X \in \text{TypeAssignment}$, we let $\mathcal{T}(X)$ denote the Type-indexed set $\{\mathcal{T}_t(X)\}_{t \in \text{Type}}$.

2. Given Type-indexed sets $M = \{M_t\}_{t \in \text{Type}}$ and $N = \{N_t\}_{t \in \text{Type}}$, a Type-indexed function $h : M \to N$ is a collection of functions $\{h_t : M_t \to N_t\}_{t \in \text{Type}}$ defined on the corresponding components of the Type-indexed sets, one for each type $t$. If $X$ is a type assignment and $M$ is a Type-indexed set, then we call the Type-indexed functions $\rho : X \to M$ $M$-environments. As usual, we let $[X \to M]$ denote the set of all $M$-environments over the assignment $X$. If $x : s \in X$ and $v \in M_s$, then we let $\rho[v/x] : X \to M$ denote the $M$-environment $\rho'$ with $\rho'(y) = \rho(y)$ for all $y \neq x$ and $\rho'(x) = v$.

   A pre-frame or pre-model is a pair $((M_t)_{t \in \text{Type}}, \{M_t^{s \to t} : M_s \times M_t \to M_t\}_{s,t \in \text{Type}})$ consisting of a Type-indexed set and a $(\text{Type} \times \text{Type})$-indexed collection of functions, such that $M_t^{s \to t}$ is extensional for any $s, t$: for any $f, g \in M_{s \to t}$, if $M_t^{s \to t}(f, v) = M_t^{s \to t}(g, v)$ for all $v \in M_s$, then $f = g$.

   A pre-frame or pre-model $M = ((M_t)_{t \in \text{Type}}, \{M_t^{s \to t} : M_s \times M_t \to M_t\}_{s,t \in \text{Type}})$ is called a frame or model of simply-typed $\lambda$-calculus iff there is a Type-indexed mapping, say $M_X$, taking well-typed $\lambda$-expressions $X \to E : t$ to mappings $M_{X \cdot E : t} : [X \to M] \to M$, with the following properties for any $M$-environment $\rho : X \to M$:

   1) $M_{X \cdot x : s}(\rho) = \rho(x : s) \in M_s$;
   2) $M_{X \cdot \lambda x : s : E}(\rho) = M_{X_s \cdot E}(\rho) \cdot x$ for any $v \in M_s$;
   3) $M_{X \cdot E : t}(\rho) = M_{X \cdot E : t}(\rho, M_{X \cdot E : t}(\rho))$.

   When such a mapping exists, we say, by a slight language abuse, that the model $M$ extends the pre-model $((M_t)_{t \in \text{Type}}, \{M_t^{s \to t} : M_s \times M_t \to M_t\}_{s,t \in \text{Type}})$.

**Exercise 209.** Show that there is at most one extension of any pre-model to a model.

(Hint: by induction on $\lambda$-expressions, using extensionality.)

Therefore, if a pre-frame can be extended to a frame, than that extension is unique. Given a model $M$ and an $M$-environment $\rho : X \to M$, we let $\rho^M : \mathcal{T}(X) \to M$ denote the Type-indexed map defined as $\rho^M(X \to E : t) = M_{X \cdot E : t}(\rho)$.

**Definition 24.** A model $M$ satisfies an equation $(\forall X) E =_t E'$, written $M \models (\forall X) E =_t E'$, iff $\rho^M(X \to E : t) = \rho^M(X \to E' : t)$ for any $\rho : X \to M$.

Given a set of equations $E$ and an equation $e$, we extend our satisfaction relation to $M \models E$ iff $M$ satisfies all equations in $E$, and $E \models e$ iff for any model $M$, if $M \models E$ then $M \models e$.

**Theorem 27.** (Soundness) If $E \vdash e$ then $E \models e$.

**Proof Sketch.** By induction on the length of the derivation. All one needs to prove is that each derivation rule is sound. For example, in the case of the $(\epsilon)$ rule, we should show that if $E \models (\forall X, x : s) E =_t E'$ then $E \models (\forall X) \lambda x : s. E =_{s \to t} \lambda x : s. E'$. Let $M$ be a model such that $M \models E$, and let $\rho : X \to M$ be an $M$-environment. Then note that

$$M^{s \to t}(\rho^M(X \to \lambda x : s. E : s \to t), v) = M_{X_s \cdot E : t}(\rho[v/x]) = M_{X_s \cdot E : t}(\rho[v/x]) = M^{s \to t}(\rho^M(X \to \lambda x : s. E' : s \to t), v).$$

343
Then by extensionality we get
\[ \rho^\#(X \to \lambda x : s.E : s \to t) = \rho^\#(X \to \lambda x : s.E' : s \to t). \]

**Exercise 210.** *Show that all the equational inference rules of simply-typed \( \lambda \)-calculus are sound.*

### 4.6.5 Full Type Frame

We next define a special (but very important) type frame, or model, of \( \lambda \)-calculus, called the **full type frame**. It consists of the most intuitive interpretation of \( \lambda \)-expressions, namely as values and functions.

Let us fix a set \( T \) and let us define inductively the following set \( \mathcal{HO} \) of sets of high-order functions starting with \( T \):

- \( T \in \mathcal{HO} \);
- \([A \to B] \in \mathcal{HO}\) whenever \( A, B \in \mathcal{HO} \).

Recall that for any two sets \( A \) and \( B \), \([A \to B]\) is the set of all functions of domain \( A \) and codomain \( B \). In other words, \( \mathcal{HO} \) is defined as the smallest set of sets that is closed under the operations above; that is, it contains \( T \) and whenever it contains the sets \( A, B \), it also contains the set of functions between them.

We can now define a unique function \( [\_] : Type \to \mathcal{HO} \) with the property that \( [\bullet] = T \) and \( [s \to t] = ([s] \to [t]) \) for any \( s, t \in Type \). Note that this function actually organizes \( \mathcal{HO} \) as a \( Type \)-indexed set: \( \mathcal{HO} = \{[t]\}_{t \in Type} \).

From now on we regard \( \mathcal{HO} \) as a \( Type \)-indexed set and organize it into a model of simply-typed \( \lambda \)-calculus. To make it a pre-model, let us define \( \mathcal{HO}^{st} : [s \to t] \times [s] \to [t] \) as expected: \( \mathcal{HO}^{st}(f, s) = f(x) \) for any \( s, t \in Type \) and any \( f : [s] \to [t] \) and \( x \in [s] \); note that \( x \) can be itself a function if \( s \) is a function type. One can immediately see that \( \mathcal{HO}^{st} \) are extensional: indeed, if \( f(x) = g(x) \) for any \( x \) then \( f = g \) (by the definition of function equality). Therefore, \( \mathcal{HO} \) is a pre-model.

To make \( \mathcal{HO} \) a model, we need to define appropriate interpretations of well-typed \( \lambda \)-expressions. For simplicity, we use the same notation \( [\_] \) as for the interpretation of types. For a given \( X \to E : t \), we define \( [X \to E : t] : [X \to \mathcal{HO}] \to [t] \) by induction as follows:

- \( [X, x : s \to x : s](\rho) \overset{def}{=} \rho(x : s) \in [s] \) for any \( \mathcal{HO} \)-environment \( \rho : X \to \mathcal{HO} \);
- \( [X \to \lambda x : s.E : s \to t](\rho)(v) \overset{def}{=} [X, x : s \to E : t](\rho)[v/x] \) for any \( \rho : X \to \mathcal{HO} \) and \( v \in [s] \);
- \( [X \to EE' : t](\rho) \overset{def}{=} ([X \to E : s \to t](\rho))( [X \to EE' : s](\rho)) \) for any \( \mathcal{HO} \)-environment \( \rho \).

**Exercise 211.** *Prove that \( \mathcal{HO} \) defined above is a model of \( \lambda \)-calculus.*

\( \mathcal{HO} \) is perhaps the most natural model of simply-typed \( \lambda \)-calculus, in which types are interpreted as sets of their corresponding values, \( \lambda \)-abstractions as functions on appropriate domains and co-domains, and \( \lambda \)-applications as function applications.

**Term model**

344
Let us now fix a Type-indexed set $X = \{X_t\}_{t \in \text{Type}}$ such that $X_t$ is infinite for any $t \in \text{Type}$ and $X_s \cap X_t = \emptyset$ for $s,t \in \text{Type}$. From now on we consider only (well-typed) $\lambda$-experiments over variables in $X$, i.e., of the form $X \triangleright E:t$ with $X \subseteq X$.

Technically speaking, since $X$ is a partition of $\bigcup_{t \in \text{Type}} X_t$, each variable is now tagged automatically with its type. This means that one can simply ignore the type assignment $X$ when writing well-typed terms $X \triangleright E:t$. However, for uniformity in notation, we prefer to keep the assignments in the notation of terms; we can think of them as the variables over which the corresponding $\lambda$-expression was intended to be defined. For example, the right-hand-side in the equation $(\forall a:s,b:s) (\lambda x:s.\lambda y:s.x)ab =_s a$ was intended to be $(a:s,b:s) \triangleright a:s$ in order for the equation to make sense, even though $b:s$ is not necessary in the type assignment.

Given a set of equations $E$, we define the $E$-equivalence class of the a $\lambda$-expression $X \triangleright E:t$ as the set

$$ [X \triangleright E:t]_E \overset{\text{def}}{=} \{ X' \triangleright E':t \mid \text{there is some } Y \text{ such that } E \vdash (\forall Y) E =_t E' \}. $$

**Proposition 18.** $[X \triangleright E:t]_E =_E [X' \triangleright E':t]_E$ iff there is some $Y$ such that $E \vdash (\forall Y) E =_t E'$.

We can now define a Type-indexed set $\mathcal{T}_E = \{\mathcal{T}_{E,t}\}_{t \in \text{Type}}$, by letting $\mathcal{T}_{E,t}$ be the set $\{ [X \triangleright E:t]_E \mid X \subseteq X \}$ for any $t \in \text{Type}$.

Further, we can extend $\mathcal{T}_E$ to a pre-model, by defining functions $\mathcal{T}_E^{\delta,t} : \mathcal{T}_{E,s \rightarrow t} \times \mathcal{T}_{E,s} \rightarrow \mathcal{T}_{E,t}$ for any types $s,t \in \text{Type}$ as follows:

$$ \mathcal{T}_E^{\delta,t}([X \triangleright E:s \rightarrow t]_E,[Y \triangleright F:s]_E) \overset{\text{def}}{=} [X \cup Y \triangleright EF:t]_E. $$

**Proposition 19.** $\mathcal{T}_E$ is a pre-model.

**Proof.** All we need to show is that $\mathcal{T}_E^{\delta,t}$ is well-defined and extensional.

For well-definedness, we need to prove that if $[X' \triangleright E' : s \rightarrow t]_E = [X \triangleright E : s \rightarrow t]_E$ and $[Y' \triangleright F' : s]_E = [Y \triangleright F : s]_E$ then $[X' \cup Y' \triangleright E'F' : t]_E = [X \cup Y \triangleright EF : t]_E$. Since there are some $\bar{X}$ and $\bar{Y}$ such that $E \vdash (\forall \bar{X}) E =_{s,t} E'$ and $E \vdash (\forall \bar{Y}) F =_s F'$, by using the rule (add) a finite number of times we can derive $E \vdash (\forall \bar{X} \cup \bar{Y}) E =_{s,t} E'$ and $E \vdash (\forall \bar{X} \cup \bar{Y}) F =_s F'$; then by (application) we can derive $E \vdash (\forall \bar{X} \cup \bar{Y}) EF =_t E'F'$. By Proposition 18 it follows that $[X \cup Y \triangleright EF : t]_E = [X \cup Y \triangleright E'F' : t]_E$. (This is the $\forall$ step.)

For extensionality, we need to show that given $X \triangleright E : s \rightarrow t$ and $X' \triangleright E' : s \rightarrow t$ such that $[X \cup Y \triangleright EF : t]_E = [X' \cup Y \triangleright E'F : t]_E$ for any $Y \triangleright F : s$, it is indeed the case that $[X \triangleright E : t]_E = [X' \triangleright E' : t]_E$. Let us pick $Y = \{y:s\} \not\subseteq X \cup X'$ and $F = y$. Then $[X,y:s \triangleright Ey : t]_E = [X',y:s \triangleright Ey : t]_E$, so by Proposition 18 $E \vdash (\forall Z) EY =_t E'Y$ for some $Z \subseteq X$. Note that, in order for $Z \triangleright Ey : t$ and $Z \triangleright E'Y : t$ to be well-typed, $Z$ must contain the variable $y : s$. Let $Z$ be $W,y:s$. By rule (δ) we then derive $E \vdash (\forall W) \lambda y : s.Ey =_{s,t} \lambda y : s.E'Y$. Finally, by applying the rule (η) twice we can derive $E \vdash (\forall W) E =_{s,t} E'$, which confirms our proof that $[X \triangleright E : t]_E = [X' \triangleright E' : t]_E$. Therefore, $\mathcal{T}_E$ is a pre-model. □

Our goal next is to organize $\mathcal{T}_E$ as a model. To do it, we first need to define mappings $\mathcal{T}_{E,X \triangleright E} : \{X \triangleright \mathcal{T}_E\} \rightarrow \mathcal{T}_{E,t}$ for all $X \triangleright E : t$. Note that $\mathcal{T}_E$-environments map variables to $E$-equivalence classes of $\lambda$-expressions. If $X = \{x_1 : s_1, \ldots, x_n : s_n\}$ and $\rho : X \rightarrow \mathcal{T}_E$ is a $\mathcal{T}_E$-environment taking $x_i$ to, say, $[X_i \triangleright E_i : s_i]_E$, then we let $\mathcal{T}_{E,X \triangleright E}(\rho)$ be defined as $[\bigcup_{i=1}^{n} X_i \triangleright E[E_1, \ldots, E_n/x_1, \ldots, x_n]]_E$, where $E[E_1, \ldots, E_n/x_1, \ldots, x_n]$ is the term obtained by substituting $E_1, \ldots, E_n$ for $x_1, \ldots, x_n$ in $E$ in parallel. One way to achieve this is to choose some fresh variables $z_1 : x_1, \ldots, z_n : x_n$ in $X \setminus (X \cup \bigcup_{i=1}^{n} X_i)$ and to let $E[E_1, \ldots, E_n/x_1, \ldots, x_n]$ be defined as $E[z_1/x_1] \ldots [z_n/x_n][E_1/z_1] \ldots [E_n/z_n]$. 345
Exercise 212. Why would it not be correct to define parallel substitution as E[E_1/x_1]...[E_n/x_n]?

Propositions 18 and 17 tell us that the maps \( T_{E,X,s} : [X \to T_E] \to T_{E,t} \) are indeed well defined.

Proposition 20. \( T_E \) is a model of simply-typed \( \lambda \)-calculus.

Proof. We need to prove that the three conditions in the definition of a model hold.

1) Let \( X : x : s \) be a type assignment and let \( \rho : X,x : s \to T_E \) be a \( T_E \)-environment where \( \rho(x_i : s_i) = [X_i \mapsto E_i : s_i]_E \) for all \( x_i : s_i \in X \) and \( \rho(x : s) = [Y \mapsto F : s]_E \). Then \( T_{E,(X,x : s)\to t}(\rho) \) is by definition \( [Y \cup \bigcup_{i=1}^n X_i \mapsto F : s]_E \), which is equal to \( \rho(x : s) \) by Proposition 18 noting that \( E \vdash (Y \cup \bigcup_{i=1}^n X_i) F =_2 F \) by (reflexivity).

2) Let \( X \mapsto \lambda x : s.E : s \to t \) be a well formed \( \lambda \)-expression, let \( \rho : X \to T_E \) be a \( T_E \)-environment, and let \( [Y \mapsto F : s]_E \) be an \( E \)-equivalence class in \( T_{E,s} \). We need to show that \( T_{E,(X,x : s)\to t}(\rho) \cdot [Y \mapsto F : s]_E = T_{E,(X,x : s)\to t}(\rho(x \leftarrow [Y \mapsto F : s]_E)) \).

If \( \rho(x_i : s_i) = [X_i \mapsto E_i : s_i]_E \) for each \( x_i : s_i \in X \), then
\[
T_{E,(X,x : s)\to t}(\rho) = \rho^b(X \mapsto \lambda x : s.E : s \to t) = \\
= \bigcup_{i=1}^n X_i \mapsto \lambda x : s.E[E_1, \ldots, E_n/x_1, \ldots, x_n] : s \to t|_E,
\]
so the left-hand-side of the equality becomes, after an application of \((\beta)\), \( [Y \cup \bigcup_{i=1}^n X_i \mapsto E[E_1, \ldots, E_n, F/x_1, \ldots, x_n, x] : t]_E \), which is by definition equal to \( [T_{E,(X,x : s)\to t}(\rho([Y \mapsto F : s]|_E)/x)] \).

3) Easy.

Exercise 213. Prove 3) above, thus completing the proof that \( T_E \) is a model.

4.6.6 Completeness

We are now ready to prove one of the most important results of simply-typed \( \lambda \)-calculus, namely the completeness of the equational deduction rules. In other words, we show that the equational rules are sufficient to derive any equation that is true in all models of \( \lambda \)-calculus.

Let us first investigate the satisfaction in \( T_E \). By definition, \( T_E \models (\forall X) E =_t E' \) iff for any \( T_E \)-environment \( \rho : X \to T_E \), it is the case that \( \rho^b(X \mapsto E : t) = \rho^b(X \mapsto E' : t) \). If \( \rho(x_i : s_i) = [X_i \mapsto E_i : s_i]_E \) for any \( x_i : s_i \in X \), then the above says that \( [\bigcup_{i=1}^n X_i \mapsto E[E_1, \ldots, E_n/x_1, \ldots, x_n] : t]_E = [\bigcup_{i=1}^n X_i \mapsto E'[E_1, \ldots, E_n/x_1, \ldots, x_n] : t]_E \), or by Proposition 18 that there is some \( Y \subseteq X \) such that \( E \vdash (Y \cup \bigcup_{i=1}^n X_i) \) \( \models (\forall Y) ) \) \( E[E_1, \ldots, E_n, F/x_1, \ldots, x_n] =_t E'[E_1, \ldots, E_n/x_1, \ldots, x_n] \). Taking \( \rho \) to be the identity \( T_E \)-environment, that is, \( \rho(x_i : s_i) = [x_i : s_i \mapsto x_i : s_i] \), we obtain that \( T_E \models (\forall X) ) \) \( E =_t E' \) implies \( E \vdash (\forall Y) ) \) \( E =_t E' \) for some \( Y \subseteq X \). By Proposition 16 we then get the following important result:

Proposition 21. \( T_E \models (\forall X) E =_t E' \) iff \( E \vdash (\forall X) E =_t E' \).

Corollary 9. \( T_E \models E \).

Theorem 28. (Completeness) If \( E \models e \) then \( E \vdash e \).

Proof. If \( E \models e \) then by Corollary 9 we get that \( T_E \models e \), so by Proposition 21 we obtain that \( E \vdash e \). \( \square \)
4.6.7 Cartesian Closed Categories as Simply-Typed Lambda-Calculus Models

We next focus on showing that CCCs can be organized as models of simply-typed \( \lambda \)-calculus. The idea is to interpret types as objects and \( \lambda \)-expressions as morphisms. As expected, exponentiation is going to play a crucial role. Indeed, if \( A \) and \( B \) are objects types interpreting \( s \) and \( t \), respectively, then \( B^A \) will interpret the type \( s \to t \).

To make our categorical framework and modeling more interesting, in the sequel we consider a common extended variant of simply typed \( \lambda \)-calculus, namely one with a set of type constants (as opposed to just one type constant, \( \circ \), like before), a product type constructor, and a set of typed \( \lambda \)-constants.

Formally, let \( S \) be a set of type constants, also possibly called basic or primitive types. The extended \( \lambda \)-calculus that we consider from here on has types built over the following grammar:

\[
\text{Type} ::= \text{S} | \text{Type} \to \text{Type} | \text{Type} \times \text{Type}
\]

As expected, types \( s \times t \) stand for pairs of values, one of type \( s \) and the other of type \( t \).

Let us also consider a set \( \Sigma \) of typed \( \lambda \)-constants, that is, pairs \( \sigma : t \), where \( t \) is some type. Like for type assignments, we assume that \( \Sigma \) does not contain constants with multiple types and let \( \Sigma(\sigma) \) denote the type \( t \) such that \( \sigma : t \in \Sigma \). We also take the liberty to write \( \sigma \in \Sigma \), without mentioning the type of \( \sigma \). Then the formal CFG of (possibly not well-typed) \( \lambda \)-expressions is:

\[
\text{Exp} ::= \text{Var} | \Sigma | \lambda x : s . \text{Exp} | \text{ExpExp} | (\text{Exp}, \text{Exp}) | \text{proj}_1 \text{Exp} | \text{proj}_2 \text{Exp}
\]

The set of constant \( \lambda \)-expressions \( \Sigma \) is also called a signature. Besides the constants in the signature, our extended simply-typed \( \lambda \)-calculus also includes the pairing construct, as well as the two corresponding projections. The typing rules for the new constructs are as follows:

\[
\begin{align*}
X \vdash \sigma : s & \quad \text{for any } X \text{ and } \sigma : s \in \Sigma; \\
X \vdash E_1 : t_1 & \quad X \vdash E_2 : t_2 \quad \text{for any } X, \text{ any Exp-terms } E_1, E_2, \text{ and any types } t_1, t_2; \\
X \vdash (E_1, E_2) : t_1 \times t_2 & \quad \text{for any type assignment } X, \text{ Exp-term } E, \text{ and types } t_1, t_2; \\
X \vdash \text{proj}_1 E : t_1 & \quad \text{for any type assignment } X, \text{ Exp-term } E, \text{ and types } t_1, t_2; \\
X \vdash \text{proj}_2 E : t_2 & \quad \text{for any type assignment } X, \text{ Exp-term } E, \text{ and types } t_1, t_2. 
\end{align*}
\]

Two equations need to be added, to capture the relationship between pairing and projections:

\[
\begin{align*}
(\text{Proj})_1 & \quad \mathcal{E} \vdash (\forall X) \text{proj}_1 (E_1, E_2) =_{t_1} E_1 \quad \text{if } X \vdash E_1 : t_1 \text{ and } X \vdash E_2 : t_2; \\
(\text{Proj})_2 & \quad \mathcal{E} \vdash (\forall X) \text{proj}_2 (E_1, E_2) =_{t_2} E_2 \quad \text{if } X \vdash E_1 : t_1 \text{ and } X \vdash E_2 : t_2.
\end{align*}
\]

Exercise 214. Show that it is not necessarily the case that \( \mathcal{E} \vdash (\forall X)(\text{proj}_1 E, \text{proj}_2 E) =_{t_1 \times t_2} E \) whenever \( X \vdash E : t_1 \times t_2 \). What if the types \( s \) associated to constants \( \sigma : s \in \Sigma \) are such that they contain no product type construct?

Simply-Typed \( \lambda \)-Calculus Captures Algebraic Specifications

347
The extension of simply-typed \(\lambda\)-calculus defined above captures algebraic signatures quite elegantly. Given a many-sorted algebraic signature \((S, \Sigma)\), i.e., a set of sorts \(S\) and a set of operations \(\sigma : s_1 \times \ldots s_n \rightarrow s\) (we define algebraic signatures in Maude, for example), all one needs to do is to declare the sorts in \(S\) as basic types and the operations in \(\Sigma\) as \(\lambda\)-constants, noting that \(s_1 \times \ldots s_n\) are types that can be constructed in \(\lambda\)-calculus (assume right or left associativity for the product of types).

One can show that the simply-typed \(\lambda\)-calculus obtained this way is a conservative extension of the original equational theory, in the sense that it can derive precisely all the equational consequences of the equational theory.

\((S, \Sigma)\)-CCC

The simply-typed \(\lambda\)-calculus defined above is therefore parameterized by the pair \((S, \Sigma)\) of basic types and constant \(\lambda\)-expressions. We next show that any CCC providing interpretations for just the basic types in \(S\) and for the \(\lambda\)-constants in \(\Sigma\) can be extended to a model of (simply-typed) \(\lambda\)-calculus, providing interpretations for all the types and for all the \(\lambda\)-expressions. This is possible because the CCCs provide elegant corresponding categorical machinery to the syntactic constructs of types and \(\lambda\)-expressions.

An \((S, \Sigma)\)-CCC is a CCC \(\text{Cxt}\) together with

- a mapping \(\llbracket \_ \rrbracket : S \rightarrow |\text{Cxt}|\), associating some object \(\llbracket s \rrbracket \in |\text{Cxt}|\) to any \(s \in S\);
- some morphism \(\llbracket \sigma \rrbracket : \ast \rightarrow \llbracket t \rrbracket\) for any \(\sigma : t \in \Sigma\), where \(\llbracket \_ \rrbracket : \text{Type} \rightarrow |\text{Cxt}|\) extends the map above as follows:
  - \(\llbracket s \rightarrow t \rrbracket \overset{\text{def}}{=} \llbracket t \rrbracket^{\llbracket s \rrbracket}\), and
  - \(\llbracket s \times t \rrbracket \overset{\text{def}}{=} \llbracket s \rrbracket \times \llbracket t \rrbracket\).

Recall that \(\ast\) is the final object of \(\text{Cxt}\).

We show that the operator \(\llbracket \_ \rrbracket : \Sigma \rightarrow \text{Cxt}\) above can be extended to any \(\lambda\)-expressions. Our goal is therefore to define for each well-typed \(\lambda\)-expression \(X \triangleright E : t\), where \(X = x_1 : s_1, \ldots, x_n : s_n\), a morphism \(\llbracket X \triangleright E : t \rrbracket : \llbracket s_1 \rrbracket \times \ldots \times \llbracket s_n \rrbracket \rightarrow \llbracket t \rrbracket\) in \(\text{Cxt}\).

Before that, let us first discuss how such morphisms would relate to the more natural interpretation of well-typed \(\lambda\)-expressions in Henkin models, namely as functions \(M_{X \triangleright E : t} : [X \rightarrow M] \rightarrow M_t\) on environments. The key observation here is the set of \(M\)-environments \([X \rightarrow M]\) is in a one-to-one correspondence with the set \(M_{s_1} \times \ldots \times M_{s_n}\). Indeed, an \(M\)-environment is nothing but a choice of an element in each of the sets \(M_{s_1}, \ldots, M_{s_n}\), which is nothing but an element in \(M_{s_1} \times \ldots \times M_{s_n}\). Therefore, we regard the functions \(M_{X \triangleright E : t}\) that appear in a Henkin model as functions \(M_{s_1} \times \ldots \times M_{s_n} \rightarrow M_t\), which now look closely related to our morphisms \(\llbracket X \triangleright E : t \rrbracket : \llbracket s_1 \rrbracket \times \ldots \times \llbracket s_n \rrbracket \rightarrow \llbracket t \rrbracket\) defined next. We will actually see later on that there is a very tight relationship between the CCC models defined next and Henkin models.

Recall that \(A \times B\) and \(B \times A\) are isomorphic in \(\text{Cxt}\). Therefore, the objects \(\llbracket s_1 \rrbracket, \ldots, \llbracket s_n \rrbracket\) can be permuted in any order in the product \(\llbracket s_1 \rrbracket \times \ldots \times \llbracket s_n \rrbracket\). However, in what follows, we prefer to work with type assignments as lists rather than sets; this way we can generate fixed interpretations rather than the possibly confusing up-to-isomorphism ones. To accomplish this, we need to slightly modify the inference rules to account for lists rather than sets. Fortunately, this is quite easy. We only need to replace each “\(X, x : s\)” by “\(X_1, x : s, X_2\)” correspondingly in the premise of rules, where \(X_1\) and \(X_2\) are any type assignments. For example, the rule (\(\xi\))
becomes

\[
\frac{E \vdash (\forall X_1, x : s, X_2) \ E =_{t} E'}{E \vdash (\forall X_1, X_2) \ \lambda x : s. E =_{s=_{t}} \ \lambda x : s. E'}
\]

Now we can extend \([\_] : Type \rightarrow |\text{Cxt}|\) to type assignments, i.e., \([\_] : TypeAssignment \rightarrow |\text{Cxt}|\), as follows:

\[
[\pi_1 : \prod_{i=1}^{n} s_i] \overset{def}{=} \left[ s_1 \right] \times \ldots \times \left[ s_n \right]
\]

We are now ready to define the morphisms \([X \triangleright E : t] : [X] \rightarrow [t]\) in \text{Cxt}, inductively on the structure of well-typed \(\lambda\)-expressions:

- \([X_1, x_i : s_i, X_2 \triangleright x_i : s_i] \overset{def}{=} \pi_i\), the \(i\)-th projection given by the definition of the product, i.e., \(\pi_i : \left[ X_1 \right] \times \left[ s_i \right] \times \left[ X_2 \right] \rightarrow \left[ s_i \right]\);

- \([X \triangleright \sigma : t] \overset{def}{=} \lambda_{|X|} : \left[ \sigma \right]\) for any \(\sigma : t \in \Sigma\), where \(\lambda_{|X|} : \left[ X \right] \rightarrow \star\) is the unique morphism from \([X]\) to the final object \(\star\), and \([\sigma] : \star \rightarrow [t]\) is the interpretation of \(\sigma\) given as part of the definition of the \((S, \Sigma)\)-CCC \text{Cxt}. Intuitively, the morphism \([X \triangleright \sigma : t]\) forgets its input and recalls the built in \(\sigma\);

- \([X \triangleright \lambda x : s. E : s \rightarrow t] \overset{def}{=} \text{curry}([X, x : s \triangleright E : t])\). Note that, indeed, this morphism is well-defined: this is because \([X, x : s \triangleright E : t]\) is a morphism \([X] \times \left[ s \right] \rightarrow [t]\) and \text{curry} takes morphisms in \text{Cxt}(\left[ X \right] \times \left[ s \right], [t]) to morphisms in \text{Cxt}(\left[ X \right], [t]^{\left[ s \right]}). Therefore, \text{curry}([X, x : s \triangleright E : t])\) is a morphism \([X] \rightarrow [s \rightarrow t]\), as expected. Note the elegance of the definition of this CCC interpretation, in contrast to the corresponding definition in Henkin models;

- \([X \triangleright E_1 E_2 : t] \overset{def}{=} \left[ [X \triangleright E_1 : s : t], [X \triangleright E_2 : s] \right]; \text{app}^{\left[ t \right]^{\left[ s \right]} \times \left[ s \right]}\). This definition needs some explanation. Note that \([X \triangleright E_1 : s \rightarrow t]\) is a morphism \([X] \rightarrow [t]^{\left[ s \right]}\), while \([X \triangleright E_2 : s]\) is a morphism \([X] \rightarrow [s]\), so \([X \triangleright E_1 : s \rightarrow t]\), \([X \triangleright E_2 : s]\) is a morphism \([X] \rightarrow [t]^{\left[ s \right]} \times \left[ s \right]\). This morphism can be indeed composed with \(\text{app}^{\left[ t \right]^{\left[ s \right]} \times \left[ s \right]} : [t]^{\left[ s \right]} \times \left[ s \right] \rightarrow [t]\), thus giving a morphism \([X] \rightarrow [t]\), as desired. All these morphisms are depicted in the following figure:

\[
\begin{align*}
&\text{app}^{\left[ t \right]^{\left[ s \right]} \times \left[ s \right]} : [t]^{\left[ s \right]} \times \left[ s \right] \rightarrow \left[ t \right] \\
&\pi_1 : \left[ X \triangleright E_1 : s \rightarrow t \right] \rightarrow \left[ t \right]^{\left[ s \right]} \\
&\pi_2 : \left[ X \triangleright E_2 : s \right] \rightarrow \left[ s \right] \\
&\left[ X \right] : \left[ X \triangleright E_1 : s \rightarrow t \right] \times \left[ X \triangleright E_2 : s \right] \rightarrow \left[ t \right] \\
&\left[ X \right] \rightarrow \left[ X \triangleright E_1 : s \rightarrow t \right] \\
&\left[ X \triangleright E_2 : s \right] \rightarrow \left[ s \right]
\end{align*}
\]

Like in the diagram above, it is often the case that constructions and proofs in category theory are driven almost automatically by diagrams; note that there are no other simpler ways to put together the morphisms \([X] \rightarrow [t]^{\left[ s \right]}\) and \([X] \rightarrow [s]\) into a morphism \([X] \rightarrow [t]\).

The remaining definitions are straightforward:

- \([X \triangleright (E_1, E_2) : t_1 \times t_2] \overset{def}{=} \left[ [X \triangleright E_1 : t_1], [X \triangleright E_2 : t_2] \right]\

- \([X \triangleright \text{proj}_1 E : t_1] \overset{def}{=} [[X \triangleright E : t_1 \times t_2]] \pi_1;\)
\[ \{x \mapsto \text{proj}_2 E \cdot t_2 \} \; \overset{\text{def}}{=} \; \{x \mapsto E \cdot t_1 \times t_2\} \cdot \pi_2. \]

We can associate a morphism \( \{x \mapsto E \cdot t\} : \{X \} \rightarrow \{T\} \) to any well-typed \( \lambda \)-expression \( X \mapsto E \cdot t \). Let us construct such a morphism for a concrete \( \lambda \)-expression:

\[
\begin{align*}
\{x \mapsto \lambda f : s : f : s \rightarrow t, g : s \times s \rightarrow s, x : (s \times s \rightarrow t) \rightarrow t, y \mapsto s \rightarrow s, X \mapsto x : s, f : s \rightarrow s, g : s \times s \rightarrow s \} &= \\
\{x \mapsto \lambda f : s : f : s \rightarrow t, g : s \times s \rightarrow s, x : (s \times s \rightarrow t) \rightarrow t, y \mapsto s \rightarrow s, X \mapsto x : s, f : s \rightarrow s, g : s \times s \rightarrow s \} &= \\
\{x \mapsto \lambda f : s : f : s \rightarrow t, g : s \times s \rightarrow s, x : (s \times s \rightarrow t) \rightarrow t, y \mapsto s \rightarrow s, X \mapsto x : s, f : s \rightarrow s, g : s \times s \rightarrow s \} &= \\
\{x \mapsto \lambda f : s : f : s \rightarrow t, g : s \times s \rightarrow s, x : (s \times s \rightarrow t) \rightarrow t, y \mapsto s \rightarrow s, X \mapsto x : s, f : s \rightarrow s, g : s \times s \rightarrow s \} &= \\
\{x \mapsto \lambda f : s : f : s \rightarrow t, g : s \times s \rightarrow s, x : (s \times s \rightarrow t) \rightarrow t, y \mapsto s \rightarrow s, X \mapsto x : s, f : s \rightarrow s, g : s \times s \rightarrow s \} &= \\
\{x \mapsto \lambda f : s : f : s \rightarrow t, g : s \times s \rightarrow s, x : (s \times s \rightarrow t) \rightarrow t, y \mapsto s \rightarrow s, X \mapsto x : s, f : s \rightarrow s, g : s \times s \rightarrow s \} &=.
\end{align*}
\]

where \( u \overset{\text{def}}{=} (s \rightarrow s) \rightarrow (s \times s \rightarrow t) \rightarrow t, v \overset{\text{def}}{=} s \rightarrow s, \) and \( X \overset{\text{def}}{=} x : s, f : s \rightarrow s, g : s \times s \rightarrow s. \)

Interestingly, note that the morphism obtained above contains no references to the variables that occur in the original \( \lambda \)-expression. It can be shown that the interpretation of \( \lambda \)-expressions into a CCC is \textit{invariant to \( \alpha \)-conversion}. To see that, let us draw the morphism above as a tree, where we write \( \lambda \) instead of \( \text{curry}, \_\_ \) instead of \( \langle , , \rangle; \text{app} \) \overset{\text{def}}{=} [ ]\rangle, \langle , , \rangle \) instead of the remaining \( \langle , \rangle \) and \( t \) instead of \( \pi_1 \) (and omit the types):

The (right) tree above suggests a representation of \( \lambda \)-expressions that is invariant to \( \alpha \)-conversion: each binding variable is replaced by a natural number, representing the number of \( \lambda \)'s occurring on the path to it; that number then replaces consistently all the bound occurrences of the variable. The corresponding lambda expression without variables obtained using this transformation is \( \lambda (\lambda (\lambda (3 (1, (2 1)) \lambda 2)). \)

**Exercise 215.** \textit{Explain why this representation is invariant to \( \alpha \)-conversion.}

The representation of \( \alpha \)-expressions above was explicitly proposed as a means to implement \( \lambda \)-calculus in 1971 by Nicholas de Bruijn.

In the same paper, de Bruijn proposed another encoding which became more popular. We do not know whether de Bruijn was influenced by the CCC interpretation of \( \lambda \)-expressions or not, but we discuss his other representation technique here.

**de Bruijn Nameless Representation of \( \lambda \)-expression**

The second and more popular representation technique of \( \lambda \)-expressions proposed by Nicholas de Bruijn in 1971 is a bottom-up version of the above representation. For the above example the tree representing the encoding is (we omit the types):

350
Exercise 216. Define the transformation above formally in Maude.

Exercise 217. Define application for the first de Bruijn encoding, in a similar style to the one above.

Extending Henkin models

We can extend the definition of Henkin models to \((S, \Sigma)\)-Henkin models. Thus, an \((S, \Sigma)\)-preframe or premodel (Henkin) consists of the following:

- a set \(M_t\) for each type \(t\);
- a \((\text{Type-indexed})\) mapping \(M_s : \Sigma \to M_t\);
- for any \(s, t \in \text{Type}\)
  - a function \(M^{st} : M_{s \to t} \times M_s \to M_t\) which is \textit{extensional}, i.e., for any \(f, g \in M_{s \to t}\), if \(M^{st}(f, u) = M^{st}(g, u)\) for all \(u \in M_s\) then \(f = g\);
  - two functions \(M_t^{sx} : M_{sx} \to M_t\) and \(M_t^{sxl} : M_{sxl} \to M_t\) which form an \textit{extensional pair}, i.e., for any \(p, q : M_{sx}\), if \(M_t^{sx}(p) = M_t^{sx}(q)\) and \(M_t^{sxl}(p) = M_t^{sxl}(q)\) then \(p = q\);

Given \(u \in M_s\) and \(v \in M_t\), when it exists we let \((u, v)\) denote the \textit{unique} element in \(M_{sx}\) such that \(M_t^{sx}(p) = u\) and \(M_t^{sxl}(p) = v\).

A well-typed \(\lambda\)-expression \(X \to E : t\) is interpreted by a mapping \(M_{X \to E} : [X \to M] \to M_t\) such that:

- \(M_{X \to E} [\alpha \sigma : \Sigma](\rho) = M_{\alpha \sigma} \in M_s\);
- \(M_{X \to E} [\alpha : \Sigma](\rho) = \rho(x : s) \in M_s\);
- \(M^{st}(M_{X \to E}[\alpha x : E : s \to t](\rho), v) = M_{X \to E}[\alpha x : E \to E](\rho[v/x])\) for any \(v \in M_s\);
• \( M_X \upharpoonright E_{1;2;3}(\rho) = M_s^{s;1}(M_X \upharpoonright E_{1;2;3}(\rho), M_X \upharpoonright E_{1;2;3}(\rho)) \);

• \( M_s^{s;1}(M_X \upharpoonright E_t; s; \xi(t)(\rho)) = M_X \upharpoonright E_t(\rho) \) and \( M_t^{t;1}(M_X \upharpoonright (E_t; s; \xi(t)(\rho)) = M_X \upharpoonright E_t(\rho) \);

• \( M_X \upharpoonright proj_1 E_t(\rho) = M_s^{s;1}(M_X \upharpoonright E; s; \xi(t)(\rho)) \);

• \( M_X \upharpoonright proj_2 E_t(\rho) = M_t^{t;1}(M_X \upharpoonright E; s; \xi(t)(\rho)) \).

**Henkin models are CCCs**

Given an \((S, \Sigma)\)-Henkin model \( M \), one can define a \((S, \Sigma)\)-CCC \( C \):

• the objects of \( C \): \( M_t \) for each \( t \in \text{Type} \);

• the set of morphisms \( C(M_s, M_t) \) is \( M_{s \rightarrow t} \);

• the identity morphism \( 1_{M_t} \) is \( M_{t \upharpoonright \lambda_{x:s} \rightarrow t} \);

• given \( f \in C(M_s, M_t) \) and \( g \in C(M_t, M_{t'}) \), define \( f; g \) as \( M_{s \rightarrow t; v; \lambda_{x:s} \rightarrow t'}(\rho) \) where \( \rho(u) = f \) and \( \rho(v) = g \);

• \( M_s \times M_t = M_s^{s;1} \) and the projections are \( M_s^{s;1} \) and \( M_t^{s;1} \);

• the exponentiation object \( M_t^{M_s} \) is \( M_{s \rightarrow t} \), and the application morphism \( \text{app}^{M_{s;1}, M_t} \) is \( M_{t \upharpoonright \lambda_{x:s} \rightarrow t; \times\langle \lambda_{x:y}, \lambda_{x:z}: \times\rightarrow t \rangle} \);

• \( \| t \| = M_t \) and \( \| \sigma \| = M_{s;1} \).

**Exercise 218.** Prove that \( C \) is indeed a \((S, \Sigma)\)-CCC.

**Some CCCs are Henkin models** Let \( C \) be an \((S, \Sigma)\)-CCC such that for each object \( A, C(\star, A) \) is a homomorphically family, that is, for any object \( B \) and any two morphisms \( f, g : A \rightarrow B \), if \( h; f = h; g \) for each \( h : \star \rightarrow A \) then \( f = g \); such a category is also called well-pointed. Then we can define a Henkin model \( M \) for \((S, \Sigma)\) as follows:

• \( M_t = C(\star, \| t \|) \) for each type \( t \);

• \( M^{s;1}_{s \rightarrow t} \times M_s \rightarrow M_t \) is given by: for any \( f : \star \rightarrow \| t \| = \| s \| \times \| t \| \) and any \( x : \star \rightarrow \| s \| \), \( M^{s;1}_{s \rightarrow t}(f, x) = \langle f, x \rangle; \text{app}_{\| s \| \times \| t \|} \);

• given \( X = x_1 : s_1, \ldots, x_n : s_n \) let \( \prod_X = \| s_1 \| \times \cdots \times \| s_n \| \). Each morphism \( h : \star \rightarrow \prod_X \) is equivalent with the tuple \( \langle h; \pi_{\| s_1 \|}, \ldots, h; \pi_{\| s_n \|} \rangle \). Thus environments \( \rho : X \rightarrow M \) and morphisms \( \star \rightarrow \prod_X \) are in bijection. Let \( \bar{\rho} : \star \rightarrow \prod_X \) denote the image of \( \rho \) through this bijection;

• For each well-typed \( \lambda \)-expression \( X \upharpoonright E : s \), and each assignment \( \rho \), let \( M_X \upharpoonright E ; s(\rho) = \bar{\rho}; \| X \upharpoonright E : s \| \).

**Exercise 219.** Show that \( M \) is indeed an \((S, \Sigma)\)-Henkin model.
4.7 Recursion and PCF

We are already familiar with various facets of recursion (in languages that we encountered, in rewriting, in lambda-calculus, etc.). The following is an interesting observation that deserves some further investigation:

The simpler the programming or specification language paradigm, the simpler the treatment of recursion.

Indeed, recursion is not even noticeable as a special feature in a rewrite engine (for example Maude) or in an imperative programming language (like C), but it requires special language support, including special syntax, for more complex languages (such as ML).

Recursion in Term Rewriting

Rewriting supports recursion naturally, without any particular technical or theoretical infrastructure. It is just us, humans, who understand a certain rewriting definition as recursive; the rewrite engine does not even need to be aware of that. Consider, for example, a rewriting implementation of the length operator on lists using Maude notation:

\[
\begin{align*}
\text{eq } & \text{length(nil)} = 0 . \\
\text{eq } & \text{length(M ; L)} = 1 + \text{length(L)} .
\end{align*}
\]

From the perspective of rewriting, there is nothing special going on here. A term \( \text{length}(L) \) for some list \( L \) is iteratively rewritten using the rules above as well as the rules of integer numbers, until it is eventually reduced to a natural number. Even though we may perceive the rewriting definition of \( \text{length} \) as recursive, the rewrite engine does nothing but match-and-apply rules until a normal form is reached, which we can interpret as the “value” obtained after “evaluating” the original expression.

However, note that the process of rewriting itself is recursive. Indeed, we can regard the process of rewriting a term \( t \), say \( \text{rewrite}(t) \), as a procedure defined recursively as follows:

\[
\text{rewrite}(t)
\]

1. find a rule \( l \rightarrow r \) such that \( l \) matches some subterm \( u \) of \( t \); if no such rule exists then return \( t \);
2. let \( \theta \) be a substitution with \( \theta(l) = u \);
3. replace \( u \) by \( \theta(r) \) in \( t \) and obtain a new term \( t' \);
4. \( \text{rewrite}(t') \)

In order for rewriting definitions to be practical, one needs to ensure that they terminate. To achieve that, one typically needs to ensure that a term is always reduced to a simpler term. This reduces to showing that any instance of the right-hand side of a rule is somehow simpler than the corresponding instance of the left-hand-side. What “simpler” means is dependent on the particular definition. In the case of \( \text{length} \), simpler means the operator is applied on a strictly smaller list in the right-hand-side term. Similar well-founded techniques are needed to prove termination of recursive programs in any programming language; the point here is, again, that rewriting definitions do not need to treat recursive definitions any differently.

It is interesting to note that the concept of “simpler term” is a semantical one - the term does not have to be simpler as a tree or w.r.t. some other immediate syntactic criterion, as shown by the following rewriting definition of bubble-sort:
eq bubbleSort(L) = if process(L) == L then L else bubble(process(L)) fi .

eq process(N ; M ; L) = if N <= M then N ; process(M ; L) else M ; process(N ; L) fi .
eq process(N) = N .
eq process(nil) = nil .

Here, the computation of bubble(L) eventually reduces to the computation of bubble(L'), where the latter is simpler than the former in the sense that L' is more sorted than L.

Note, however, that one can speculate the recursive nature of rewriting and define bubble sort by just one simple and elegant conditional rewriting rule:

ceq M ; N = N ; M if N <= M .

Now there is no explicit term in the right-hand-side of the rule that one can show smaller that a corresponding term in the left-hand-side. Nevertheless, one can show that any application of the rule above would decrease the “weight” of any list that it applies on.

Besides termination, confluence is another important factor that needs to be considered when designing recursive rewriting definitions. Confluence generalizes the Church-Rosser result (for variants of λ-calculus) to any rewrite system: a rewrite system is confluent if and only if for any two reductions of a term t to t₁ and t₂, there is some term t' such that t₁ and t₂ reduce to t'. A rewrite system that is both confluent and terminating is called canonical. Note that terms have unique normal forms when reduced using canonical rewrite systems. In our context, a recursive rewriting definitions that is a canonical rewrite system can and should be regarded as a well-behaved definition.

Consider, for example, the following (deliberately more complicated than needed) definition of addition of natural numbers:

eq M + 0 = M .
eq 0 + N = N .
eq s(M) + N = s(M + N) .
eq M + s(N) = s(M + N) .

The above simulates a non-deterministic recursive definition. According to this definition, there are many ways to compute the sum of m and n, since each time there is no unique way to reduce a term containing + to a simpler one. One can relatively easily show that this definition is well-behaved, because each rule simplifies the term to reduce in some way (first two eliminate a + operator, while the other two push an s operator higher).

We can therefore see that canonical rewrite systems support in a simple and natural way arbitrarily complex recursive definitions. Moreover, since rewrite rules can be applied in parallel when their applications do not overlap, canonical rewrite systems can also be regarded as potentially very efficient implementations of recursive definitions. In particular, devise-and-concur recursive definitions can be regarded operationally as starting a new thread solving the subproblems; on a parallel implementation of a rewrite engine, one can achieve a computational speed close to as many times as processors available at no additional programming effort.

One should not underestimate the difficulty of showing rewrite systems canonical. Since rewriting is Turing-complete, not even termination is decidable. One can show that in general neither termination nor
confluence of a rewrite system is decidable. To have a better feel for the difficulty of the confluence problem, consider the following problem.

Suppose that a group of children have a large collection of black and white balls and decide to play a game. They put a bunch of balls in a bag and then each of them, at any moment they wish and potentially in parallel, extracts two balls from the bag and puts one back as follows: if the two balls have the same color they put a black ball back in the bag, otherwise they put a white one. If the bag is declared as an associative binary operator \( \cdot \cdot \), then this game can be easily defined as a four rule ground rewrite system:

\[
\begin{align*}
\cdot \cdot \rightarrow \cdot , & \quad o \cdot \rightarrow \cdot , \\
\cdot o \rightarrow o , & \quad o \cdot \rightarrow o .
\end{align*}
\]

Using variables, one can define it as just one rule:

\[
eq X Y = \text{if } X == Y \text{ then black else white fi .}
\]

**Exercise 220.** Prove that the rewrite system above is canonical.

**Recursion in Simple Imperative Languages**

Recall the implementation of the Hanoi tower problem in a simple C-like imperative language:

```c
function h(x, y, z, n) {
    if n >= 1 then {
        h(x, z, y, n - 1) ;
        print(x) ;
        print(z) ;
        h(y, x, z, n - 1)
    }
}
```

In this recursive function, \( h \) refers to itself directly and transparently. This is possible because of the default design conventions of the imperative language:

- all functions are declared at the top level of the program (more precisely at the beginning);
- no nested function declarations allowed;
- function declarations are assumed visible anywhere in the program, including in the body of the functions themselves.

Therefore, languages can support recursion naturally if they are constrained by design appropriately. Nevertheless, many programmers may not be happy with such language constraints just for the sake of supporting recursion transparently.

**Recursion in Higher Order Languages**

Recursion can be supported in a more flexible (but also more intricate) way when we climb to higher order languages, where functions, as first-order volatile citizens of the program, start raising non-trivial scoping and visibility issues. For instance, in the recursive definition of factorial

```latex
\texttt{letrec } f \texttt{ n } = \texttt{if } n \texttt{ eq } \texttt{0} \texttt{ then } 1 \texttt{ else } n \texttt{ * f(n - 1) in } f 7
\texttt{355}
```
one has to explicitly state, using letrec instead of let, that the expression defining f is not a usual, but a recursive binding. Any other trick of simulating recursion would be unnatural and statically untypeable, like the classical

\[
\text{let } f \ n \ g = \text{if } n \ \text{eq} \ 0 \ \text{then } 1 \ \text{else } n \ * \ g \ (n - 1) \ g \ \text{in } f \ 7 \ f
\]

Recursion in Untyped $\lambda$-Calculus

The latter (untypeable) definition of factorial, which did not have to use the explicit recursion construct letrec, was given in the style of untyped $\lambda$-calculus, by passing a function to itself as an argument. As we have already seen, this technique can be applied very generally in the context of untyped $\lambda$-calculus, taking advantage of the fixed point theorem:

**Theorem.** If $Y$ denotes $\lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$, then $YF \equiv \beta F(YF)$ for any $\lambda$-expression $F$.

A $\lambda$-expression $Y$ with the property above, namely that $YF \equiv \beta F(YF)$ for any $\lambda$-expression $F$, is called a fixed point operator. The particular $Y$ above is also known as the call-by-name fixed point operator (for reasons that will become clear shortly). As expected, there can are many other fixed point operators, including the following:

**Exercise 221.** Show that the following $\lambda$-expressions are also fixed point operators:

- $(\lambda x. \lambda f. f(xx))(\lambda x. \lambda f. f(xx))$, also known as the Turing fixed point operator;
- $\lambda f. (\lambda x. f(\lambda y. xxy))(\lambda x. f(\lambda y. xxy))$, also known as the call-by-value fixed point operator.

We have already discussed how the fixed-point operator $Y$ in the fixed-point theorem can be used to prove equational properties about recursive programs. For example, if we let $F$ be the usual factorial $\lambda$-expression

$$
\lambda f. \lambda n. \ \text{if } n \ \text{eq} \ 0 \ \text{then } 1 \ \text{else } n \ * \ f(\text{pred } n),
$$

where we assume if-then-else and pred, the latter taking any strictly positive number $n$ into $n - 1$ and 0 into 0, as already defined syntactic sugar (note that if-then-else has no associated evaluation strategy.), then one can easily show by using the equational rules of $\lambda$-calculus properties like $(YF)^3 \equiv \beta (F(YF))^3 \ldots$.

But how does (recursive) computation actually take place in $\lambda$-calculus? Of course, our computational mechanism must be based on $\beta$-reduction here. We have no need, and no intuitive computational meaning, for the $\eta$ rule; the latter makes sense only for expressing and proving equivalence of programs.

However, one cannot blindly apply $\beta$-reduction whenever a possibility is found inside the current term, since this way one might miss opportunities of termination; although it is true that, by the Church-Rosser theorem, no $\beta$-reduction is an irreparable mistake, nevertheless a persistent chain of wrong $\beta$-reductions might indefinitely continue a virtually terminating computation. For example, one can easily see that in the case of our factorial, the following can take place

$$(YF)^3 \Rightarrow \beta YF3 \Rightarrow \beta FYF3 \Rightarrow \beta F(FYF)3 \Rightarrow \beta F(F(F(YF)))3 \ldots,$$

where $YF$ is the $\lambda$-expression $Y[F/f]$.

**Exercise 222.** If for any $\lambda$-expression $F$ we let $YF$ denote the $\lambda$-expression $(\lambda x. F(xx))(\lambda x. F(xx))$, show that $YF \Rightarrow \beta FYF$. The fixed-point theorem tells us that $YF \equiv \beta F(YF)$ anyway; is it also true that $YF \Rightarrow \beta F(YF)$? How about the other two fixed-point operators in Exercise 221?
Thanks to the Church-Rosser theorem, we know that if a normal form exists then it is unique. In particular, 6 is the only normal form of \((YF)^3\) despite the fact that the reduction of \((YF)^3\) may not terminate. Needless to mention that, in practical implementations of programming languages, one would like to devise reduction or evaluation strategies that would lead to normal forms whenever they exist.

One simplistic way to achieve the desired computation would be by breadth-first search in the space of computations, until a \(\beta\)-normal form is eventually reached. Clearly, such a computation strategy would be unacceptably inefficient in practice. For that reason, strategies that are not based on search are desirable. Many reduction strategies have been investigated in the literature and implemented in various languages.

We next discuss one interesting reduction strategy, called \textit{leftmost-outermost} \(\beta\)-reduction, also known as \textit{call-by-name} or \textit{normal-order}: each time, \(\beta\)-reduction is applied outside in and left-to-right; this corresponds to \(\beta\)-reducing the first suitable subterm found by a prefix traversal of the term to reduce. The application of a \(\beta\)-reduction step can enable another reduction which occurs earlier in the prefix traversal, so each time a reduction is applied, the entire term to rewrite is assumed to be retraversed using the prefix strategy (better algorithms are used in practical implementations). Notice that argument expressions passed to \(\lambda\)-abstractions are not evaluated at \textit{call time}, but instead at \textit{need time}.

The first programming language providing a call-by-name evaluation strategy was Algol-60. An important variation of call-by-name, supported by several programming languages including most notoriously Haskell, is \textit{call-by-need}. Under call-by-need, \(\lambda\)-expressions are evaluated at most once: any subsequent use of a \(\lambda\)-expression just uses the previously calculated normal form. As we know from CS422, in the context of side effects call-by-name and call-by-need can lead to different normal forms.

\textbf{Theorem.} If a \(\lambda\)-expression has a \(\beta\)-normal form, then it can be reached by leftmost-outermost \(\beta\)-reductions.

An intuitive argument for the above result is the following: if one applies \(\beta\)-reductions as close to the top as possible, then one can prevent any unnecessary non-termination obtained by infinitely rewriting a term which is supposed to disappear anyway by an upper \(\beta\)-reduction.

\textbf{Exercise 223.} \textit{Modify your Maude implementation of \(\lambda\)-calculus (either the one based on the De Bruijn transformation or the one based on combinators) to support the call-by-name reduction strategy.} \textbf{Hint.} You only need to add appropriate rewriting strategies to certain operations in the signature.

Another very important, if not the most important, reduction strategy is the one called \textit{call-by-value}. Under call-by-value, \(\lambda\)-expressions are always evaluated \textit{before} they are passed to \(\lambda\)-abstractions. In order for call-by-value to work, one needs to inhibit reductions under certain language constructs, such as within the body of \(\lambda\)-abstractions and within the two choices of the conditional. Also, an appropriate fixed-point operator needs to be chosen.

\textbf{Exercise 224.} \textit{Reduce \((YF)^3\) using a call-by-value strategy. Can you just use the call-by-need fixed-point operator? How about the other two fixed-point operators in Exercise 221?}

\textbf{Mu Simply-Typed Lambda Calculus}

Once one decides that types are a useful feature of one’s language and one decides to declare and/or check them statically, the next important thing one should ask oneself is how such a decision affects recursive definitions.

Typed functional languages do not allow functions to pass themselves to themselves, so in order to support recursion, one needs to introduce a new language construct (\texttt{letrec}). The goal of this topic is to discuss in more depth the relationship between types and recursive definitions of functions. To focus on the
essence of the problem, we start with the simplest typed language discussed so far, namely simply-typed $\lambda$-calculus.

Let us first understand why the presented fixed-point technique to support recursive definitions in untyped $\lambda$-calculus does not work in the context of types. Consider the fixed-point combinator $Y$ suggested by the fixed-point theorem, $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. To make $Y$ a well-typed $\lambda$-expression in simply typed $\lambda$-calculus, we need to come up with some types $t_f$, $t_1^f$, and $t_2^f$, such that the $\lambda$-expression $\lambda f: t_f.(\lambda x: t_1^f.f(xx))(\lambda x: t_2^f.f(xx))$ can be typed. The reason for that is the subexpression $xx$: there is no type that one can assign to $x$ such that $xx$ can be calculated a type.

Let us assume though, for the sake of the discussion, that fixed-point combinators $Y$ existed in simply-typed $\lambda$-calculus and let us try to understand what their type should be. Recall that such combinators are intended to apply on $\lambda$-expressions $F$ encoding the one-step unrolling of a recursive definition.

For example, the unrolling $\lambda$-expression associated to the definition of the (typed) factorial function of type $\text{nat} \rightarrow \text{nat}$ would be

$$F := \lambda f:\text{nat} \rightarrow \text{nat}. \lambda n:\text{nat}. \text{if } n \text{ eq } 0 \text{ then } 1 \text{ else } n \ast f(\text{pred } n)$$

having the type $(\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat}$.

Any recursive function, of any type, has such a one-step unrolling $\lambda$-expression associated to it. For example, a function of type $\text{nat} \rightarrow \text{bool}$ returning the parity of a natural number is associated the following $\lambda$-expression of type $(\text{nat} \rightarrow \text{bool}) \rightarrow \text{nat} \rightarrow \text{bool}$:

$$F := \lambda f:\text{nat} \rightarrow \text{bool}. \lambda n:\text{nat}. \text{if } n \text{ eq } 0 \text{ then } \text{true} \text{ else } \text{not } (f(\text{pred } n)).$$

It is interesting to note that recursion can mean more than just a recursive function. For example, in lazy functional programming it is customary to define infinite data structures, such as infinite lists or infinite trees, by defining their actual behavior when asked to produce values. For example, an infinite list of zeros should be able to produce a zero whenever asked for its first element (by passing it to $\text{car}$). The unrolling $\lambda$-expression for such a recursive data-structure would be

$$F := \lambda l:\text{list} \text{ nat}. \text{cons } 0 l,$$

which has the type $\text{list} \text{ nat} \rightarrow \text{list} \text{ nat}$.

Finally, one may want to compactly define error values for any type $t$ as recursive expressions referring just to themselves, having therefore associated the trivial unrolling $\lambda$-expression $F := \lambda x:t.x$ of type $t \rightarrow t$.

Consequently, for any type $t$, it is reasonable to assume that any recursively defined $\lambda$-expression $E$ of type $t$ has an unrolling $\lambda$-expression $F_E$ of type $t \rightarrow t$. Since one expects that $E \equiv YF_E$, one can intuitively claim that a fixed-point combinator $Y$, if it exists, must have the type $(t \rightarrow t) \rightarrow t$. More precisely, one would expect for each type $t$ some corresponding fixed-point combinator $Y_t$ of type $(t \rightarrow t) \rightarrow t$.

In fact, one can prove that there exist no such magic fixed-point combinators $Y_t$ in the simply-typed $\lambda$-calculus language. It is relatively usual in computer science foundational works to enrich a certain formalism with desired features by definition, or by axiom, when those features cannot be explained or simulated by the existing framework. Since the concept of fixed-point seems to very naturally capture the essence of recursion, the solution that has been adopted by scientists to allow recursion in typed programming environments is to extend the language by simply assuming a $Y_t$ for each type $t$.

Thus, for each type $t$ we assume the existence of a new constant $Y_t:(t \rightarrow t) \rightarrow t$, called for simplicity the fixed-point operator for type $t$. Whether $Y_t$ can or not be explained from more basic principles does not concern us here. Instead, we axiomatize its behavior by requiring that

358
\[ Y_t F = F(Y_t F) \text{ for any } \lambda\text{-expression } F \text{ of type } t \rightarrow t. \]

Alternatively and more compactly, noticing that the considered expression \( F \) always has the form \( \lambda x : t. E \), we can define a new recursive binding operator \( \mu \) to bind \( x \) in \( E \) and replace the functionality of \( Y_t \) as follows:

\[ Y_t F = Y_t(\lambda x : t. E) = \mu x : t. E. \]

More precisely, we are going to have a new binding operator \( \mu \) with the same syntax as \( \lambda \) and a new equational rule saying that

\[ \mu x : t. E \equiv E[(\mu x : t. E)/x], \]

with the intuition that arbitrarily many copies of \( \mu x : t. E \) can be generated, typically by need, nesting the original expression in all places where \( x \) occurs free. An even more intuitive equivalent form of this rule is

\[ \mu x : t. E \equiv (\lambda x : t. E)(\mu x : t. E), \]

which says that \( \mu x : t. E \) is a fixed point of the function \( \lambda \)-abstracting the expression \( E \) in the variable \( x \).

So far, we only justified the \( \mu \)-extension of simply-typed \( \lambda \)-calculus by examples. A rigorously formulated reason for extending simply-typed \( \lambda \)-calculus is that it is not expressive enough as a model of computation: it can only encode very simple programs which do not have any recursive behavior. In particular, simply-typed \( \lambda \)-calculus is \textit{decidable}; this follows from two facts:

- like for untyped \( \lambda \)-calculus, the Church-Rosser theorem holds;
- unlike for untyped \( \lambda \)-calculus, all computations eventually terminate, thus leading to the reach of unique normal forms.

Hence one can decide if two terms are provably equal by simply reducing them into their normal forms and then syntactically comparing the results.

### Programming Language for Computable Functions

As one might expect, simply typed \( \lambda \)-calculus extended with the recursion operator \( \mu \) has full computational power. We are going to present a language called PCF (abbreviating Programming language for Computable Functions), a \( \mu \)-extension of simply typed \( \lambda \)-calculus with a particular but important signature \((S, \Sigma)\) together with corresponding equations.

The BNF syntax for types and expressions in PCF is the following

\[
\begin{align*}
\text{Var} & ::= x \mid y \mid \ldots \\
\text{Type} & ::= \mathbf{S} \mid \text{Type} \rightarrow \text{Type} \\
\text{Exp} & ::= \Sigma \mid \text{Var} \mid \text{ExpExp} \mid \lambda \text{Var.Type.Exp} \mid \mu \text{Var.Type.Exp}
\end{align*}
\]

where \( S = \{\text{nat, bool}\} \) is the set of basic types, and \( \Sigma = \{0 : \text{nat}, \text{succ} : \text{nat} \rightarrow \text{nat}, \text{pred} : \text{nat} \rightarrow \text{nat}, \text{true} : \text{bool}, \text{false} : \text{bool}, \text{zero?} : \text{nat} \rightarrow \text{bool}\} \cup \{\text{cond}, : \text{bool} \rightarrow t \rightarrow t \rightarrow t \mid \forall t \in \text{Type}\} \) is the signature.

When discussing typing and equations, we build on top of the already defined \((S, \Sigma)\)-simply-typed \( \lambda \)-calculus.

We only add one typing rule:

\[
\begin{align*}
X, x : t & \vdash E : t \quad \text{for any type assignment } X, x : t, \\
X & \vdash \mu x : t. E : t \quad \lambda\text{-expression } E \text{ of type } t;
\end{align*}
\]
Exercise 225. Type the following expressions:

- \( + := \mu f : \text{nat} \to \text{nat} \to \text{nat}. \lambda x : \text{nat}. \lambda y : \text{nat}. \text{condnat} (\text{zero?} \ x) \ y \ (\text{succ} \ f \ (\text{pred} \ x) \ y) \),

- \( \ast := \mu f : \text{nat} \to \text{nat} \to \text{nat}. \lambda x : \text{nat}. \lambda y : \text{nat}. \text{condnat} (\text{zero?} \ x) \ 0 \ (+ \ (f \ (\text{pred} \ x) \ y) \ y) \).  

We add the following equational rules, describing the desired behavior of the \( \Sigma \)-constants and of the recursion operator \( \mu \); by a language abuse, we let \( n \) denote \( \text{succ} \ \text{succ} \ldots \text{succ} \ 0 \) \((n \text{ times})\):

\[ \text{E} \vdash (\forall \emptyset) \text{pred} \ 0 =_{\text{nat}} 0 \]

\[ \text{E} \vdash (\forall X) \text{pred} \ \text{succ} \ n =_{\text{nat}} n \]

\[ \text{E} \vdash (\forall \emptyset) \text{zero?} \ 0 =_{\text{bool}} \text{true} \]

\[ \text{E} \vdash (\forall \emptyset) \text{zero?} \ \text{succ} \ n =_{\text{bool}} \text{false} \]

\[ \text{E} \vdash (\forall X) \text{cond} \text{t} \ \text{true} \ E \ E' =_{t} E \quad \text{if} \ X \triangleright E : t, \ X \triangleright E' : t \]

\[ \text{E} \vdash (\forall X) \text{cond} \text{f} \ \text{false} \ E \ E' =_{t} E' \quad \text{if} \ X \triangleright E : t, \ X \triangleright E' : t \]

\[ (\xi_{\mu}) \quad \text{E} \vdash (\forall X, x : t) \ E =_{t} E' \quad \text{E} \vdash (\forall X) \mu x : t. E =_{t} \mu x : t. E' \]

\[ (\mu) \quad \text{E} \vdash (\forall X) \mu x : t. E =_{t} E[(\mu x : t. E)/x] \]

\[ (\text{Con}_{\sigma}) \quad \text{E} \vdash (\forall X) \ E =_{\text{nat}} E' \quad \text{E} \vdash (\forall X) \ E' =_{\text{nat}} \sigma E' \quad \text{where} \ \sigma \in \{\text{succ, pred}\} \]

\[ (\text{Con}_{\text{zero?}}) \quad \text{E} \vdash (\forall X) \ E =_{\text{nat}} E' \quad \text{E} \vdash (\forall \emptyset) \text{zero?} \ E =_{\text{bool}} \text{false} \]

\[ (\text{Con}_{\text{cond}}) \quad \text{E} \vdash (\forall X) C =_{\text{bool}} D, \ \text{E} \vdash (\forall X) E =_{t} F, \ \text{E} \vdash (\forall X) E' =_{t} F' \quad \text{E} \vdash (\forall X) \text{cond} \text{t} E E' =_{t} \text{cond} \text{f} D F F' \]

Exercise 226. Derive the following identities:

- \( (\forall \emptyset) \ + \ (\text{succ} \ 0) \ (\text{succ} \ (\text{succ} \ 0)) =_{\text{nat}} \text{succ} \ (\text{succ} \ (\text{succ} \ 0)) \).

- \( (\forall \emptyset) \text{zero?} \ (\text{succ} \ 0) =_{\text{bool}} \text{false} \).

We have already discussed (in a rather informal context) a possible computational semantics of untyped \( \lambda \)-calculus, based on leftmost-outermost \( \beta \)-reduction.

Now we shall give the PCF language two operational semantics, a transitional or small-step one, and a natural or big-step one.
4.7.1 Transitional semantics

The transitional semantics will be given as a restricted rewriting relation on $\lambda$-terms (“restricted” in the sense that it cannot be applied inside any arbitrary context). This relation will tell us how to evaluate programs (that is, terms) to values (that is, terms which are normal forms). But what are the values in our framework? They are just the usual natural and Boolean values, plus the $\lambda$-abstractions. Here is the BNF description of values:

\[
\text{Value ::= } 0 \mid \text{succ Value} \mid \text{true} \mid \text{false} \mid \lambda \text{Var:Type.Exp}
\]

The fact that we take all $\lambda$-abstractions as values, even though it might be possible to further evaluate their body, might seem strange at a first glance; however, this is natural if we consider the body of a $\lambda$-abstraction as just the text describing a function, which will be evaluated only if the function is to be applied (remember closures from functional languages). We shall soon come back to this discussion.

Like in any functional language, we have to make a choice between several evaluation strategies when we define semantics. We choose call-by-name below, since this is immediately related to the intuition of $\beta$-reduction, but one can easily adapt it to consider other evaluation strategies.

**Exercise 227.** In a similar manner to the call-by-name transitional semantics below, give a transitional semantics to PCF following the call-by-value evaluation strategy.

We define the one-step-transition binary relation $\rightarrow$ on $\lambda$-expressions as the smallest relation closed under the following rules, where we assume that all (outermost) expressions that appear in the left-hand-side of the rules are well-typed (e.g., we assume that $FE$ in the application rule is well-typed, hence we also implicitly assume that $E$ and $F$ are typeable to some types $t \rightarrow s$ and $t$, respectively):

\[
\begin{align*}
\text{pred } 0 & \rightarrow 0 \\
\text{pred succ } V & \rightarrow V, \text{ if } V \text{ is a value} \\
\text{zero? } 0 & \rightarrow \text{true} \\
\text{zero? succ } V & \rightarrow \text{false}, \text{ if } V \text{ is a value} \\
\text{cond, true } E & \rightarrow E' \\
\text{cond, false } E & \rightarrow E' \\
E & \rightarrow E' \\
\sigma E & \rightarrow \sigma E', \text{ if } \sigma \in \{\text{succ, pred, zero}\} \\
C & \rightarrow C' \\
\text{cond, } CEE' & \rightarrow \text{cond, } C'EE' \\
F & \rightarrow F' \\
FE & \rightarrow F'E \\
(\lambda x : t.E) & \rightarrow E[E'/x] \\
(\mu x : t.E) & \rightarrow E[(\mu x : t.E)/x]
\end{align*}
\]

Note that, unlike in the case of general term rewriting, the above rules cannot be applied inside any context (subterm), but actually apply on top. Some incursions into the to-be-rewritten term context are possible, but only those obeying some restrictions:
• never evaluate the passed argument (call-by-name),
• never evaluate the second and third arguments of a condition,
• never apply $\beta$-reduction or $\mu$-unwinding in the body of another $\lambda$ or $\mu$-abstraction, etc.

And all the above negative facts are just implicit in the rules, which do not allow arbitrary congruence.

A property that one wants to be satisfied by the one-step-transition relation is that the latter is sound, in the sense that it rewrites a program into an equivalent program.

**Proposition 22.** (Soundness of the one-step transition relation) Suppose $X \triangleright E : t$. If $E \rightarrow E'$, then $\vdash (\forall X)E =_{t} E'$.

(Proof hint: Define the relation $R$ on terms as $E \equiv E'$ iff there exists a type assignment $X$ such that $\vdash (\forall X)E =_{t} E'$, and show that $\equiv$ is closed under all the transition rules.)

An important consequence is the fact that types are preserved by transitions, which allows one to type-check the program only once, at the beginning, and then not worry about types during the execution:

**Corollary 10.** If $X \triangleright E : t$ and $E \rightarrow E'$, then $X \triangleright E' : t$.

Another fact to notice is that we were able to define our values (i.e. normal forms) a priori, and then guide our notion of computation towards them. And indeed, what we initially called “values” are precisely those closed terms that cannot be further reduced:

**Proposition 23.** The following are equivalent for any closed (i.e., without free variables) expression $E$:

• $E$ is a value;
• there is no $E'$ such that $E \rightarrow E'$.

(Proof hint: Show that, on the one hand, a value cannot be rewritten and, on the other hand, that any closed non-value can be rewritten.)

Notice that there are non-closed expressions, like the variables or applications between two variables, that are neither values nor can transition in one step into anything. Moreover, there are closed expressions that are neither values nor can transition in one step into any expression other than themselves, like, for instance, $\mu x : t . x$.

**Exercise 228.** Characterize the non-closed expressions satisfying the former property. Is there any other expression, not $\alpha$-equivalent to $\mu x : t . x$, satisfying the latter property?

Define $\rightarrow^*$, the transition relation, to be the reflexive and transitive closure of $\rightarrow$. By the soundness of the one-step-transition relation, one can immediately infer

**Theorem** (Soundness of the transition relation) Suppose $X \triangleright E : t$. If $E \rightarrow^* E'$, then $\vdash (\forall X)E =_{t} E'$.

**Corollary 11.** (Subject reduction) If $X \triangleright E : t$ and $E \rightarrow^* E'$, then $X \triangleright E' : t$.

Since our language is deterministic, any expression can evaluate to at most one value:

**Proposition 24.** If $E$ is an expression and $V, V'$ are values such that $E \rightarrow^* V$ and $E \rightarrow^* V'$, then $V = V'$. 362
Proof hint: First show that, for each expression $E$, there exists at most one expression $E'$ such that $E \rightarrow E'$.

If $E \rightarrow^* V$, then we say that $E$ evaluates to $V$. There are expressions (like $\mu x : t. x$) that do not evaluate to any value; this is natural, since not all programs terminate (on all inputs). But if an expression $E$ evaluates to some value $V$, then this unique value is considered to be the result of the computation. Thus we have a partial function mapping expressions to values, which could be regarded as the final product of the transitional semantics.

What about completeness?

We saw that the transition relation is sound w.r.t. equivalence of programs. But how complete is it? That is, what can we infer about equivalence of two programs by evaluating them transitionally? Well, according to this restricted and pragmatic form of evaluation, not much. And here are two reasons for that (can you find more reasons?):

- $\eta$-equivalence is not in the scope of our transitional semantics; for instance, $\lambda f : nat \rightarrow nat. \lambda y : nat. f y$ and $\lambda f : nat \rightarrow nat. f$ are different values;

- even without the $\eta$ equation, two different values can be proved equivalent, if they are $\lambda$-abstractions; for instance, $\lambda x : nat.(\lambda y : nat.y)x$ and $\lambda x : nat.x$; this is actually a semantical decision, since we do not want to look into the body of functions, unless we have to apply them.

However, one can prove a restricted form of completeness:

**Theorem (Basic-type-completeness of the transition)** If $\vdash (\forall X) E =_t E'$ and $t \in \{nat, bool\}$, then, for each value $V$, $E$ evaluates to $V$ if and only if $E'$ evaluates to $V$.

(Proof hint: First show that, if $V$ is a value of type $t \in \{nat, bool\}$ and $\vdash (\forall X) E =_t V$, then $E \rightarrow^* V$.)

Hence, if we restrict our class to programs of basic types only, two programs that are provably equivalent either both do not terminate, or they both terminate and return the same value. Note that the restriction of returning basic typed values does not inhibit higher-order programming: diverse functions can be passed as arguments and returned as values in the process of calculating an integer value! Note that the restriction to basic types is crucial.

4.7.2 Natural Semantics

The two-step process of first defining a transition on expressions, and then extracting from it a partial map from expressions to values, can be performed in only one step. The natural semantics captures directly the idea of evaluation $E \rightarrow^* V$, by giving rules in terms of pairs (Expression,Value). The binary relation $\Downarrow$ between expressions and values is defined as the smallest closed under the following rules (below, $V$ denotes a value, and $E, E_1, E_2$ expressions; obviously, $\text{succ } V$ is then also a value; we assume that all expressions below are well-typed:}

363
\[
\begin{array}{ll}
V \Downarrow V & E \Downarrow V \\
\text{pred } E \Downarrow 0 & \text{succ } E \Downarrow \text{succ } V \\
E \Downarrow 0 & \text{zero? } E \Downarrow \text{true} \\
\text{pred } E \Downarrow 0 & \text{zero? } E \Downarrow \text{false} \\
E \Downarrow \text{false} & \text{cond } C E_1 E_2 \Downarrow V \\
\text{cond } C E_1 E_2 \Downarrow V & E_1 \Downarrow \lambda x : s.E , E[E_2/x] \Downarrow V \\
E_1 E_2 \Downarrow V & \mu x : t.E \Downarrow V
\end{array}
\]

The above rules describe the evaluation process in a bottom-up fashion, starting from the values. One can show that transitional and natural semantics are equivalent:

**Proposition 25.** For each expression \( E \) and value \( V \), \( E \Downarrow V \) iff \( E \rightarrow^* V \).

The next corollary, expressing the determinism of evaluation, might be instructive to be proved directly about natural semantics:

**Corollary 12.** If \( E \) is an expression and \( V, V' \) are values such that \( E \Downarrow V \) and \( E \Downarrow V' \), then \( V = V' \).

### 4.7.3 Denotational Semantics of PCF

We are now ready to define a canonical model of PCF. Unlike the canonical functional models of simply-typed \( \lambda \)-calculus, our model should be able to capture non-termination of programs; non-termination will be handled by the interpretation of the recursion operator \( \mu \). In our semantical framework,

- basic types will be interpreted as certain CPOs with bottom element \( \perp \), where \( \perp \) will stand for “undefined”;
- types will be interpreted as continuous functions between the interpretations of their component types;
- environments will be, as usual, \((\text{Type-indexed})\) mappings from type assignments into the model;
- well-typed expressions will be interpreted as mappings from environments into the interpretation of their types.

In what follows, we consider *bottomed CPOs* (CPOs for short), i.e., structures of the form \((P, \leq, \perp)\), where \((P, \leq)\) is a CPO and \( \perp \) is its bottom element. Note that \( \text{Cont}((P, \leq, \perp), (Q, \leq, \perp_Q)) \), the set of continuous functions from \( P \) to \( Q \), can be naturally endowed with a CPO structure, if we let \( f \leq g \) iff \( f(p) \leq g(p) \) for all \( p \in P \); the bottom element will be the function \( \perp \) defined by \( \perp(p) = \perp_Q \) for all \( p \in P \). In what follows, this will be the CPO structure that we will implicitly assume on \( \text{Cont}((P, \leq, \perp), (Q, \leq, \perp_Q)) \).

**Exercise 229.** Prove that \( \text{Cont}((P, \leq, \perp_p), (Q, \leq, \perp_Q)) \), with the indicated structure, is indeed a CPO.

**Exercise 230.** Prove that CPOs, together with continuous functions between them, form a category.
Recall that \( \text{Nat} \) and \( \text{Bool} \) denote the sets of natural numbers and Booleans respectively. Let \( \perp \notin \text{Nat} \cup \text{Bool} \), and let \( \text{Nat}_\perp \) and \( \text{Bool}_\perp \) be the sets \( \text{Nat} \cup \{\perp\} \) and \( \text{Bool} \cup \{\perp\} \) respectively, endowed with the partial orders \( \{(\perp, i) \mid i \in \text{Nat}\} \cup \{(i, i) \mid i \in \text{Nat} \cup \{\perp\}\} \) and \( \{(\perp, \text{false}), (\perp, \text{true}), (\perp, \perp), (\text{true}, \text{true}), (\text{false}, \text{false})\} \) respectively.

**Exercise 231.** Find a natural correspondence between the set of partial functions from \( X \) to \( Y \) and the set of continuous functions from \( X_\perp \) to \( Y_\perp \), where \( X, Y \in \{\text{Nat}, \text{Bool}\} \). Is this correspondence bijective?

**Interpreting types**

We are now ready to define \( \sem{} : \text{Type} \to \{\text{The class of CPOs}\}: \)

- \( \sem{\text{nat}} = \text{Nat}_\perp \), \( \sem{\text{bool}} = \text{Bool}_\perp \); 
- \( \sem{s \to t} = \text{Cont}(\sem{s}, \sem{t}) \).

We let \( \mathcal{HO}_\perp \) denote the Type-indexed set \( \{\sem{s}\}_{s \in \text{Type}} \).

**Interpreting constants**

- \( \sem{\text{succ}}, \sem{\text{pred}} \in \{\text{nat} \to \text{nat}\} = \text{Cont}(\text{Nat}_{\perp}, \text{Nat}_{\perp}) \), \( \sem{\text{succ}}(v) = \begin{cases} v + 1 & \text{if } n \in \text{Nat} \\ \perp & \text{if } v = \perp \end{cases} \), \( \sem{\text{pred}}(v) = \begin{cases} v - 1 & \text{if } n \in \text{Nat} - \{0\} \\ 0 & \text{if } n = 0 \\ \perp & \text{if } v = \perp \end{cases} \)
- \( \sem{\text{true}}, \sem{\text{false}} \in \{\text{bool}\}\), \( \sem{\text{true}} = \text{true}, \sem{\text{false}} = \text{false} \)
- \( \sem{\text{cond}} \in \{\text{bool} \to \text{t} \to \text{t} \to \text{t}\} = \text{Cont}(\text{Cont}(\sem{t}, \sem{t})), \sem{\text{cond}}(b)(v_1)(v_2) = \begin{cases} v_1 & \text{if } b = \text{true} \\ v_2 & \text{if } b = \text{false} \\ \perp & \text{if } b = \perp \end{cases} \)

**Exercise 232.** Show that all the above are correct interpretations, in the sense that \( \sem{\text{succ}}, \sem{\text{pred}}, \sem{\text{cond}}, \sem{\text{cond}}(b), \sem{\text{cond}}(b)(v) \) are indeed continuous functions.

**Interpreting well-typed terms**

We define \( \sem{X : E : t} : \{X \to \mathcal{HO}_{\perp}\} \to \sem{t} \) recursively on the structure of \( X : E : t \).

- \( \sem{X : x : t}(\rho) = \rho(x) \) if \( x \) is a variable;
- \( \sem{X : \sigma : t}(\rho) = \sem{\sigma} \in \sem{t} \) if \( \sigma \) is a constant;
- \( \sem{X \to E_1 E_2 : t}(\rho) = (\sem{X \to E_1 : s \to t}(\rho))(\sem{X \to E_2 : s}(\rho)) \), if \( X \to E_1 : s \to t \) and \( X \to E_2 : s \);
- \( \sem{X \to \lambda x : s : E : s : t}(v) = \sem{X, x : s \to E : t}(\rho[v/x]) \) for each \( v \in \sem{s} \);
- \( \sem{X \to \mu x : t : E : t}(\rho) = \text{fix}(\sem{\lambda x : t : E : t \to t})(\rho) \).

**Exercise 233.** Show that the above mapping is correctly defined, in the sense that the returned values indeed belong to the specified codomains. (Hint: The proof will proceed, as usually, by induction on \( E \); note that there is nothing to prove for the case of a \( \mu \)-operator on top.)
Let us try to explain the intuition behind the definition of \( [X \triangleright \mu x : t. E : t] (\rho) \). To parallel the syntactic intuition of \( \mu \) (given formally in the equational proof system which acts like a guide for all our semantic frameworks), the desired denotation would be a fixed point of the function whose law is expressed by \( E \), that is, a fixed point of \( [\lambda x : t. E : t](\rho) \) (since we want to be able to unwind \( \mu x : t. E \) into \( E(\mu x : t. E)/x \), i.e., into \( (\lambda x : t. E) \mu x : t. E \)). But why the least fixed point? Intuitively, we do not want \( [X \triangleright \mu x : t. E : t](\rho) \) to possess more information than the one provided by iterated unwindings.

**Exercise 234.** The denotation of \( X \triangleright \mu x : t. E : t \) could be equivalently expressed as \( \text{fix}(g) \), where \( g : [t] \rightarrow [t] \) is defined by \( g(v) = [X, x : t \triangleright E : t](\rho[v/x]) \) for all \( v \in [t] \).

As usually, we say that \( \mathcal{HO}_\bot \) satisfies an equation \( \forall X \ E =_t E' \), denoted \( \mathcal{HO}_\bot \models \forall X \ E =_t E' \), iff \( [X \triangleright E : t] = [X \triangleright E' : t] \).

**Theorem** \( \mathcal{HO}_\bot \) is a model of PCF, in the sense that it satisfies all the PCF rules.

**Corollary 13.** If PCF \( \vdash \forall X \ E =_t E' \), then \( [X \triangleright E : t] = [X \triangleright E' : t] \).

**Corollary 14.** If \( E \triangleright E', X \triangleright E : t \) and \( X \triangleright E' : t \), then \( [X \triangleright E : t] = [X \triangleright E' : t] \).

**Corollary 15.** If \( E \downarrow V, X \triangleright E : t \) and \( X \triangleright V : t \), then \( [X \triangleright V : t] \neq \bot \).

**Proposition 26.** If \( V \) is a value and \( X \triangleright V : t \), where \( t \in \{\text{nat, bool}\} \), then \( [X \triangleright V : t] \neq \bot \).

**Corollary** If \( X \triangleright E : t, X \triangleright V : t \) and \( [X \triangleright E : t] = [X \triangleright V : t] \), then \( E \downarrow V \).

### 4.8 Polymorphisms

**Types of Polymorphism**

The word *polymorphism* comes from the Greek language, where it means “having many forms”. In the study of programming languages, people use polymorphism to state that the same expression or operation or segment of code can be used with different types. Generally, there are three kinds of polymorphism in programming languages:

- **Overloading or coercion.** For example, we can use + on related entities, such as integers, natural numbers, and/or real numbers, as well as on unrelated ones, such as strings and/or Booleans. However, typically the underlying algorithms are entirely different; think, for example, of addition on integers versus addition on float number. Many algebraic specification languages, including Maude, support operator overloading. In combination with subsorting, as most of you are already aware, operator overloading can lead to quite non-trivial theoretical and practical issues. We do not intend to discuss this kind of polymorphism in this class.

- **Parametric/universal polymorphism.** This kind of polymorphism, also encountered under the name of “generics” in the literature, will be discussed in depth shortly. The idea underlying parametric, or universal, polymorphism is that types need not be concrete, but rather have parameters which can be instantiated by need. While parametric types can be quite handy in many situations, they may lead to difficult technical problems, especially in the context of type inference.

- **Subtype polymorphism.** This is mostly found and considered indispensable in object-oriented languages, because of the intrinsic need of these languages to support inheritance. We will also discuss subtype polymorphism in this course. As it is customary in discussions on subtyping, we will set up a relatively simple formal framework, extending the simply typed \( \lambda \)-calculus with records.
Parametric/Universal Polymorphism

To capture the essence of parametric polymorphism in a general and uniform way, we next consider an extension of typed $\lambda$-calculus with universal types. This extension is known in the literature under the names polymorphic $\lambda$-calculus, second-order $\lambda$-calculus, or system F, and represents the foundation for many works in type theory.

Interestingly, this calculus was invented independently by two famous scientists, the logician Jean-Yves Girard and the computer scientist John Reynolds.

The extra ingredients in this calculus are type variables, which can be universally quantified, and type instantiations.

The BNF syntax for types and expressions is extended as follows (as usual, we use the color red for the new features):

- $\text{TypeVar} ::= s | t | \ldots$
- $\text{Type} ::= S | \text{Type} \rightarrow \text{Type} | (\forall \text{TypeVar})\text{Type}$
  (where $S$ is some set of basic (constant) types)
- $\text{Var} ::= x | y | \ldots$
- $\text{Exp} ::= \text{Var} | \text{ExpExp} | \lambda \text{Var}.\text{Exp} | \lambda \text{TypeVar}.\text{Exp} | \text{Exp}[\text{Type}]$

Type variables $s, t, \ldots$, etc., will be used essentially as parameters for universal types. These should not be confused with the basic types $S$ (such as bool, int, etc.).

A universal type is quantified universally by a type variable, with the intuition that it represents a collection of types, one for each particular instance of the parameter. For example, $(\forall t)t \rightarrow t$ represents the universal type of the (polymorphic) identity function: it can be regarded as a collection of functions, one for each instance of the type $t$.

Besides the usual $\lambda$-expressions, we now allow type-parametric expressions together with a means to instantiate them. Precisely, a $\lambda$-expression $\lambda t. E$ represents the $\lambda$-expression $E$ parameterized by the type $t$; that means that $E$ can freely use the type $t$ just like any other type constant (those in $S$), but, however, when required by an instantiation, say $(\lambda t. E)[T]$ where $T$ is any type, one should be able to replace each free occurrence of $t$ in $E$ by $T$. The meaning of parametric types will be formally given as usual with equations.

In this enriched context, type assignments need to be extended appropriately to consist of not only typed variables of the form $x : T$, but also of type variables of the form $t$:

- $\text{TypeAssignment} ::= \emptyset | \text{Var}:\text{Type}, \text{TypeAssignment} | \text{TypeVar}, \text{TypeAssignment}$

In what follows, we let $E, E', \ldots$ denote expressions, $T, T', \ldots$ types and $X, X', X_1, X_2, \ldots$ type assignments.

Typing Rules

For deriving the well-formed expressions $X \triangleright E : t$, we consider all the previous typing rules, plus:

- $X, x : T \triangleright x : T$ if all the free type variables in $T$ are in $X$
- $X \triangleright \lambda t. E : (\forall t)T$
- $X \triangleright E : (\forall t)T$
- $X \triangleright E[T'] : T'[t/t]$ if all the free type variables in $T'$ are in $X$
It is intuitively clear what the “free type variables in a type” are—those that are not bound by any universal quantifier; also $T'[T/t]$ is the type obtained from $T'$ by replacing each free occurrence of $t$ in $T'$ with $T$. Note that, like in the case of substitution in $\lambda$-calculus, some renamings of bound variables might be necessary in order to avoid type variable captures. All these can be formally expressed:

**Exercise 235.** Define formally $\text{Free}(T)$ and $T[T'/t]$.

**Exercise 236.** Define a type checker for System F equationally, in Maude. Your type checker should take a closed term $E$ as input and return a type $T$ if and only if $\emptyset \Rightarrow E : T$ (otherwise it can return any error type—the “core dump” thing).

**Equational Rules**

We consider all the previous equational rules that we defined for $\lambda$-calculus, plus the following three rules giving the expected meaning to the new features. The first two rules are congruence- or ($\xi$)-like rules for the new syntax, while the third gives meaning to type instantiations:

\[
(\forall X, t) E =_T E' \\
(\forall X) \lambda t.E =_{(\forall)T} \lambda t.E' \\
(\forall X) E =_{(\forall)T} E' \\
(\forall X) E[T'] =_{T[T'/t]} E'[T'] \text{ if } \text{Free}(T') \subseteq X
\]

\[
(\forall X) (\lambda t.E)[T'] =_{T[T'/t]} E[T'/t] \text{ if } X, t : E : T \text{ and } \text{Free}(T') \subseteq X
\]

All the equations that can be derived using the rules above are well-typed:

**Proposition 27.** If $(\forall X) E =_T E'$ is derivable with the rules above then $X \Rightarrow E : T$ and $X \Rightarrow E' : T$.

Carrying and checking the type of equalities can be quite inconvenient in efforts to efficiently automate the applications of the equational rules above (this would be more or less equivalent to dynamic type checking). A common practice in formal definitions of typed languages is, whenever possible, to drop the subscript types of equalities and to derive instead untyped equations. The three rules above would then be:

\[
(\forall X, t) E = E' \\
(\forall X) \lambda t.E = \lambda t.E' \\
(\forall X) E = E' \\
(\forall X) E[T'] = E'[T'] \text{ if } \text{Free}(T') \subseteq X
\]

\[
(\forall X) (\lambda t.E)[T'] = E[T'/t] \text{ if } \text{Free}(T') \subseteq X
\]

When type subscripts are dropped from equations, a natural question arises: are the equations consistent with the types? While the execution of the rules above becomes much simplified, the problem with dropping the types is that one could be able to derive equalities containing expressions which are not well-typed:

**Exercise 237.** Give two examples of such meaningless equalities that can be derived with the untyped equational rules above. The two examples should reflect two different problems of the resulting deduction system (more precisely, of the last two rules above).

Fortunately, the untyped equations preserve the well-typed-ness, which is the main result supporting and justifying type-checking:
Proposition 28. (Type preservation) If $(\forall X) \ E = E'$ is derivable with the untyped equational rules and $X \triangleright E : T$ for some type $T$, then $X \triangleright E' : T$.

In practical implementations of programming languages, the result above says that if one wants to correctly execute a program $E$, all one needs to do is to type-check $E$ before execution and then ignore the types during the execution as far as the equational rules above are not violated. This allows more efficient implementations, which is precisely why most compilers have built-in type-checkers as front-ends.

Different transitional semantics can be now given by orienting and restricting the applications of the equations above accordingly. Since the obtained transition relations are included in the equational derivation relation, that is, $(\forall X) \ E \rightarrow E'$ implies $(\forall X) \ E = E'$, the type preservation property also holds for the various transitional semantics.

Notice that polymorphic $\lambda$-calculus is, in some sense, an untyped (w.r.t. type variables) $\lambda$-calculus over a typed (w.r.t. usual variables) $\lambda$-calculus. For that reason, polymorphic $\lambda$-calculus is also often called second-order typed $\lambda$-calculus. Instead of going through the theoretical intricacies of this calculus, we shall just give some examples showing how it can be used in parameterized programming.

**Some Examples**

**Polymorphic conditional.**

Recall that in simply typed $\lambda$-calculus we had a conditional constant for any type $t$, namely $\text{cond}_t : \text{bool} \rightarrow t \rightarrow t \rightarrow t$. In polymorphic $\lambda$-calculus we can instead define just one constant of polymorphic type, namely $\text{cond} : (\forall t) \text{bool} \rightarrow t \rightarrow t \rightarrow t$. Given a type $T$, it follows by the typing rules that $\text{cond}[T]$ has the type $\text{bool} \rightarrow T \rightarrow T \rightarrow T$. To capture the expected meaning of the conditional, two equational rules need to be added (we only consider the untyped equations here):

$$(\forall X) \ \text{cond}[T] \ \text{true} \ E \ E' = E \text{ if } \text{Free}(T) \subseteq X$$

$$(\forall X) \ \text{cond}[T] \ \text{false} \ E \ E' = E' \text{ if } \text{Free}(T) \subseteq X$$

**Exercise 238.** Does the type preservation property still hold when the polymorphic conditional is added to the language? If yes, prove it. If no, give a counter-example.

The drawback of polymorphic conditionals in particular, and of expressions of polymorphic type in general, is that one needs to instantiate them explicitly whenever one wants to apply them.

It is interesting to note that the conditional in Maude is also polymorphic (type “show module TRUTH .”), but that one does not need to instantiate it explicitly.

**Exercise 239.** Would it be possible to change the definition of polymorphic $\lambda$-calculus so that one does not need to instantiate polymorphic expressions explicitly, that is, so that polymorphic expressions are instantiated automatically depending on the context in which they are used? Comment on the advantages and the disadvantages of such a language.

**Identity function.**

Recall the FUN example that we used to motivate the concept of let-polymorphism:

```
let i = fun x -> x
in if (i true) then (i 7) else 0
```

Without let-polymorphism the program above will not type, that is, there is no way to find a type - by type inference or otherwise - to $i$, making useless the polymorphic declarations of functions.
However, in polymorphic $\lambda$-calculus, one can define the identity function explicitly polymorphic (or universal) as the $\lambda$-expression $\lambda t. \lambda x : t . x$ of universal type $(\forall t) t \to t$. Then the FUN program above can be given as the following $\lambda$-expression which is well-typed (has the type nat) in polymorphic $\lambda$-calculus:

$$ (\lambda i : (\forall t) t \to t . \text{cond[nat]} (i[[\text{bool}]] \text{true}) (i[[\text{nat}]] 7) 0) (\lambda t. \lambda x : t . x) $$

**Function composition.**

Function composition is a standard operation in functional programming languages. One would like to generically compose two functions $f : t_1 \to t_2$ and $g : t_2 \to t_3$ to obtain a function $t_1 \to t_3$, for any types $t_1$, $t_2$ and $t_3$. The corresponding $\lambda$-expression in polymorphic $\lambda$-calculus is

$$ \lambda t_1. \lambda t_2. \lambda t_3. \lambda f : t_1 \to t_2. \lambda g : t_2 \to t_3. \lambda x : t_1. g(f x) $$

and will type to $(\forall t_1)(\forall t_2)(\forall t_3)(t_1 \to t_2) \to (t_2 \to t_3) \to t_1 \to t_3$.

**Exercise 240.** Derive the type of the function composition expression formally, using the typing rules.

**Exercise 241.** Would it make any sense to introduce the parameter types by need, that is, to define function composition as

$$ \lambda t_1. \lambda t_2. \lambda f : t_1 \to t_2. \lambda t_3. \lambda g : t_2 \to t_3. \lambda x : t_1. g(f x) $$

Type this expression and comment on its advantages and disadvantages in contrast to the standard polymorphic function composition above.

**On Recursion**

Recall that in simply-typed $\lambda$-calculus there was no way to type an expression of the form $\lambda x : T. xx$, for any type $T$. Let us try to represent this expression within System F.

First, note that $\lambda x : (\forall t) t \to t. xx$ is not a good choice, because it would not type. Indeed, trying to type it, we need to find some type $T$ such that $x : (\forall t) t \to t \to xx : T$; then the type of the entire expression would be $(\forall t) t \to t \to T$. Trying to type the expression $xx$, one gets the type constraint $(\forall t) t \to t \to T = (\forall t) t \to t$, which, unfortunately, does not admit a solution even in the enriched context of polymorphic types.

The solution is to use a type instantiation in order to change the type of the first $x$ to one which can be applied on the second $x$: the expression $\lambda x : (\forall t) t \to t . x[(\forall t) t \to t] x$, say $E$, will type to $(\forall t) t \to t \to t$.

Note, however, that we cannot type anything like similar to the untyped $\omega$ combinator, namely $(\lambda x.xx)(\lambda x.xx)$, or to the untyped fixed-point operators. As a matter of fact, we cannot simulate recursion in System F without $\mu$.

**Exercise 242.** Try, and fail, to simulate recursion in system F. You should see that, despite the fact that one may use instantiation to type expressions previously untypable in simply typed $\lambda$-calculus, one actually cannot use polymorphism for dealing with recursion.

For that reason, like we did in simply typed $\lambda$-calculus, to deal with recursion we extend our calculus with the usual $\mu$-abstraction. Assume the same usual typing/equational rules for $\mu$.

**More Examples**

**Polymorphic lists.**
Recall that one of the major problems in typing FUN programs in CS422 was the fact that lists were not polymorphic. Because of that reason, we were not able to regard, for example, the empty list as both a list of integers and as a list of Booleans.

Polymorphic λ-calculus, fortunately, supports polymorphic lists very naturally. In what follows we add the list type construct, give the signature of the usual list operators, and show how polymorphic lists can be used in practice. Here we are not concerned with how lists are represented, nor with how the list operators are defined: we just assume them given. We will later discuss Church lists, showing that all the list operators can be defined from more basic principles.

Let us first add the type construct for lists:

\[ Type = \ldots \mid \text{list Type} \]

The usual operators on lists can be now given as just constants of universal types:

\[
\begin{align*}
\text{nil} & : (\forall t) \text{list } t \\
\text{cons} & : (\forall t) t \to \text{list } t \to \text{list } t \\
\text{nil?} & : (\forall t) \text{list } t \to \text{bool} \\
\text{car} & : (\forall t) \text{list } t \to t \\
\text{cdr} & : (\forall t) \text{list } t \to \text{list } t
\end{align*}
\]

Supposing these list operations already defined, let us define a polymorphic \textit{map} operator, which takes as arguments a list and a function and applies that function on each element of the list:

\[
\lambda t_1 \cdot \lambda t_2 \cdot \lambda f : t_1 \to t_2 \cdot \\
\mu m : \text{list } t_1 \to \text{list } t_2 \cdot \\
\lambda l : \text{list } t_1 . \\
\text{cond}[\text{list } t_2] (\text{nil?[}t_1\text{] } l) \\
\quad (\text{nil}[t_2]) \\
\quad (\text{cons } [t_2] (f (\text{car}[t_1] l)) (m (\text{cdr}[t_1] l)))
\]

\textbf{Exercise 243.} Derive the type of the map expression above using the formal type inference rules.

\textbf{Exercise 244.} Define and type a reverse function on polymorphic lists.

\textit{Church Booleans.}

Recall the Church Booleans from untyped λ-calculus. They already had a polymorphic intuitive meaning: \textit{true} and \textit{false} were the first and the second projection functions, respectively, expecting two arguments of the same type. With the machinery of polymorphic λ-calculus, we can take the Boolean type to be the following universal type:

\[
\text{bool}_1 := (\forall t) t \to t \to t
\]

and the two Boolean constants to be:

\[
\begin{align*}
\text{true} & := \lambda t . \lambda x : t . \lambda y : t . x \\
\text{false} & := \lambda t . \lambda x : t . \lambda y : t . y
\end{align*}
\]
Note that the two expressions above have indeed the type $\text{bool}_A$. The logical operators can now be naturally defined. For instance, not can be defined as $\lambda b : \text{bool}_A. \lambda t. \lambda x : t. \lambda y : t [b] y x$.

Exercise 245. Define and type the other Church Boolean operators.

Church numerals.
Recall that, in untyped $\lambda$-calculus, Church numerals were characterized by their latent application to a successor function $s$ and a zero element $z$, which would result in applying $s$ to $z$ a certain number of times. Therefore, the expected type of $s$ would be $t \rightarrow t$, where $t$ is the type of $z$. Thus we can define the polymorphic type of Church numerals as

$$\text{nat}_A := (\forall t) (t \rightarrow t) \rightarrow t \rightarrow t$$

Number $n$ can then be encoded as the expression of type $\text{nat}_A$

$$n_A := \lambda t. \lambda s : t \rightarrow t. \lambda z : t. s(s \ldots (s z) \ldots),$$

with $n$ applications of $s$. All the usual arithmetical operations on numbers can now be defined. For instance,

$$\text{succ}_A := \lambda n : \text{nat}_A. \lambda t. \lambda s : t \rightarrow t. \lambda z : t. s (n[t] s z) +_A := \lambda n : \text{nat}_A. \lambda m : \text{nat}_A. \lambda t. \lambda s : t \rightarrow t. \lambda z : t. n[t] s (m[t] s z).$$

Exercise 246. Define the polymorphic multiplication and power.

Church lists.
Following a similar idea as for Church numerals, one can define Church lists. A list is regarded through its latent application to a “binary operation” $f$ and an “initial value” $v$, which would result in iteratively applying the binary operation as follows: first to the last element of the list and $v$, then to the last but one element and the previous result, and so on, until the list is exhausted. For instance, the list $[a, b, c]$ is encoded as $\lambda f. \lambda v. f a (f b (f c v))$.

For each type $t$, let us define the polymorphic type

$$\text{list}_t := (\forall p) (t \rightarrow p \rightarrow p) \rightarrow p \rightarrow p$$

Note, however, that we want the list operations to be themselves polymorphic. For example, we want the type of cons to be $(\forall t) t \rightarrow \text{list}_t \rightarrow \text{list}_t$. We can now define the usual list operators quite naturally:

$$\text{nil} := \lambda t. \lambda p. \lambda f : t \rightarrow p \rightarrow p. \lambda v : p. v$$

(the type of nil is $(\forall t) \text{list}_t$),

$$\text{cons} := \lambda t. \lambda \text{head} : t. \lambda \text{tail} : \text{list}_t.$$

$$\lambda p. \lambda f : t \rightarrow p \rightarrow p. \lambda v : p.$$

$$f \text{ head} (\text{tail}[p] f v)$$

(the type of cons is $(\forall t) t \rightarrow \text{list}_t \rightarrow \text{list}_t$),

$$\text{nil?} := \lambda t. \lambda l : \text{list}_t. l[\text{bool}_A] (\lambda x : t. \lambda y : \text{bool}_A. \text{false}) \text{ true}$$

(the type of nil? is $(\forall t) \text{list}_t \rightarrow \text{bool}_A$),

$$\text{car} := \lambda t. \lambda l : \text{list}_t. l[t] (\lambda x : t. \lambda y : t. x) (\text{error}[t])$$

(the type of car is $(\forall t) \text{list}_t \rightarrow t$)
Note that car is intended to be a partial function, undefined on the empty list; hence we introduced a polymorphic constant error, of type \((\forall t)t\). This constant is also useful for any other partial function definitions. Some prefer to define “error for type \(t\)” as an infinite recursion, for example \(\mu x : t . x\); in other words, they either replace each instance error[\(t\)] by \(\mu x : t . x\), or otherwise add an equation error = \(\lambda t . \mu x : t . x\). The drawback of this adhoc convention is that the evaluation of expressions applying car will not terminate under call-by-value evaluation strategy.

**Exercise 247.** Define and formally type the cdr operator.

**Giving System F an Executable Semantics**

Let us now focus on the original formulation of System F, namely the one without recursion (\(\mu\)). Let us orient all the equations left-to-right, thus obtaining a transitional semantics \((\rightarrow)\) of System F. One of the most famous results of System F is the following:

**Theorem (Girard).** (very hard) In System F without \(\mu\), the transition relation \((\rightarrow)\) is confluent and terminates.

**Exercise 248.** Define System F equationally in Maude. Your definition should take a polymorphic \(\lambda\)-expression and evaluate it to its unique normal form.

**Hint.** You may need to use the de Bruijn technique to properly avoid variable captures; note that variable captures can now appear both at the \(\lambda\)-expression level and at the type level. Do not worry whether your normal form contains de Bruijn indexes.

**Type inference/reconstruction** The problem of type inference can be stated in the polymorphic \(\lambda\)-calculus framework as follows.

Given any untyped \(\lambda\)-expression \(E\), is it the case that \(E\) is typable in System F?

In other words, is there any polymorphic \(\lambda\)-expression \(E_F\) in System F such that erase(\(E_F\)) = \(E\)? Here, the operator erase just loses any type information and can be formally defined as follows:

\[
\begin{align*}
\text{erase}(x) &= x, \\
\text{erase}(\lambda x : T . E) &= \lambda x . \text{erase}(E), \\
\text{erase}(E_1 E_2) &= \text{erase}(E_1) \text{erase}(E_2), \\
\text{erase}(\lambda t . E) &= \text{erase}(E), \\
\text{erase}(E[T]) &= \text{erase}(E)
\end{align*}
\]

This problem, open for more than 20 years, was finally proven to be undecidable. Algorithms were developed to partially solve this problem, some of them based on Huet’s partial algorithms for higher-order unification (another undecidable problem).

**Subtype Polymorphism**

We next focus on the other major approach to polymorphism, namely subtype polymorphism, which is mostly encountered in the context of modern object-oriented programming languages.

To stay focused on the major aspects of subtype polymorphism, we here introduce a very simply \(\lambda\)-calculus language extended with various features that reflect most of the interesting issues related to subtype polymorphism encountered in other languages or formalisms.

**Simply Typed \(\lambda\)-Calculus with Records**

We extend simply typed \(\lambda\)-Calculus by adding records. Let Field be a countably infinite set, disjoint from Var, and let us extend types and expressions as follows:
Thus a record type is a set of typed attributes, each represented as a \((\text{Field}, \text{Type})\) pair. For example, we can declare a record type, say \(\text{person}\), as \(\{\text{name} : \text{string}, \text{age} : \text{nat}, \text{height} : \text{nat}, \ldots\}\). Two kinds of expressions are introduced for dealing with records:

- **record expressions** (written like sets of equalities, assigning expressions to the fields of a record), and
- **field accessing** expressions.

The first one creates a record by assigning a value to every attribute, while the second fetches the value of an attribute of a record. For instance, the expression

\[
\{\text{name} = \text{‘John Smith’}, \text{age} = 27, \text{height} = 180, \ldots\}
\]

defines a record variable, say \(\text{john}\). We can get the values of its fields with expressions like \(\text{john.name}\), \(\text{john.age}\), etc.

Records inherently bring the issue of **subtyping**. Intuitively, if a context requires a record \(R\) of a certain kind, it should be the case that a record \(R’\) having more information than needed be still suitable for that context. Thus we would like to be able to write

\[
(\lambda p : \{\text{age} : \text{nat}\}. (p.\text{age}))
\]

\[
\{\text{name} = \text{‘John Smith’}, \text{age} = 27, \text{height} = 180, \ldots\}
\]

However, with the previous typing system, this is not allowed since the argument type is not exactly the same as the parameter type. To avoid this kind of unnecessary constraints, but still maintain a rigorous typing discipline, we introduce the following important notion of subtyping, first intuitively and then rigorously:

**Intuitive definition of subtyping:** Given two types \(t_1\) and \(t_2\), we say that \(t_1\) is a subtype of \(t_2\), written \(t_1 \leq t_2\), iff \(t_1\) has at least the same information as \(t_2\), or, in other words, a value of type \(t_1\) can be used wherever a value of type \(t_2\) is expected; one also may say that \(t_1\) is more concrete than \(t_2\), or that \(t_2\) is more general than \(t_1\).

**Subtyping Rules**

Now we set up the rules that will allow us to formally derive subtype relationships of the form \(t_1 \leq t_2\), which will be further needed to define the typing system:

\[
\begin{align*}
\text{\(\leq\)-reflexivity} & \quad t \leq t \\
\text{\(\leq\)-transitivity} & \quad \frac{t_1 \leq t \quad t \leq t_2}{t_1 \leq t_2} \\
\text{\(\leq\)-arrow} & \quad \frac{t_2 \leq t_1 \quad t'_1 \leq t'_2}{t_1 \rightarrow t'_1 \leq t_2 \rightarrow t'_2} \\
\text{\(\leq\)-record} & \quad \frac{t_1 \leq t'_1 \quad \ldots \quad t_m \leq t'_m}{[l_1 : t_1, \ldots, l_n : t_n] \leq [l_1 : t'_1, \ldots, l_m : t'_m]} \text{ if } m \leq n
\end{align*}
\]

The first two rules are clear. To understand the \(\leq\)-arrow rule, let us suppose a context which expects a value of type \(t_2 \rightarrow t'_2\) but actually receives a value \(V\) of type \(t_1 \rightarrow t'_1\). In a presumptive later use, \(V\) might
be applied to an argument $W$ of type $t_2$. To assure that $V$ can handle $W$, one needs that every inquiry that $V$ makes to its argument (expected to be of type $t_1$) be answered by $W$, i.e., that $W$ provides at least as much information as a value of type $t_1$; thus $t_2 \leq t_1$. Furthermore, the result of applying $V$ to $W$ is a value of type $t'_1$, while a value of type $t'_2$ would be expected; thus $t'_1 \leq t'_2$.

The rule (\text{-record}) says two important things: first, that the subtype record must include the fields of the supertype record, and second, that the types of those fields in the subtype record are subtypes of the types of those fields in the supertype record. Both these facts are needed in order for a value of subtype record to be used in contexts where values of supertype record are expected.

The next two properties of the subtype relation say that one can only derive meaningful subtypings. These properties may be used later to prove important properties of type systems supporting subtyping.

\textbf{Exercise 249.} If $t \leq t_2 \rightarrow t'_2$ then $t$ has the form $t_1 \rightarrow t'_1$ such that $t_2 \leq t_1$ and $t'_1 \leq t'_2$.

\textbf{Exercise 250.} If $t \leq \{l_1 : t'_1, \ldots, l_m : t'_m\}$ then $t$ has the form $\{l_1 : t_1, \ldots, l_n : t_n\}$ with $m \leq n$ and $t_i \leq t'_i$, $i \in \{1, \ldots, m\}$.

\textbf{Type System}

Building upon the subtype relation formally defined above, we can now give subtype-flexible rules for deriving types:

\textbf{(subsumption)} \quad \frac{X \triangleright E : t_1}{X \triangleright E : t_2} \quad \text{if } t_1 \leq t_2

\textbf{(record)} \quad \frac{X \triangleright E_1 : t_1 \ldots X \triangleright E_n : t_n}{X \triangleright \{l_1 = E_1, \ldots, l_n = E_n\} : \{l_1 : t_1, \ldots, l_n : t_n\}}

\textbf{(field access)} \quad \frac{X \triangleright E : \{l_1 : t_1, \ldots, l_n : t_n\}}{X \triangleright E.l_j : t_j} \quad \text{if } j \in \{1, \ldots, n\}

(subsumption) allows us to lift the type of an expression to any supertype, thus formally justifying our informal claim that “expressions of type $t_1$ can be used in any context where expressions of supertype $t_2$ are expected”.

(record) allows us to derive a type of a record from the names and the types of its fields, while (field access) allows us to derive the type of a field once the type of the entire record is known.

\textbf{Proposition 29.} If $X \triangleright (\lambda x : t.E) : t_1 \rightarrow t_2$ then $t_1 \leq t$ and $X, x : t \triangleright E : t_2$.

\textbf{Proposition 30. (Substitution.)} If $X, x : s \triangleright E : t$ and $X \triangleright F : s$ then $X \triangleright E[x \leftarrow F] : t$.

\textbf{Equational Rules}

The following natural equational rules are added to those of simply-typed \textlambda-calculus. We here assume the untyped variants of equational rules, which, as usual, will rise the question of type preservation:

$$(\forall X) \{l_1 = E_1, \ldots, l_n = E_n\}, l_i = E_i \text{ for all } i \in \overline{1,n}$$

$$(\forall X) E = E' \text{ for any field } l$$

$$(\forall X) E.l = E'.l$$

$$(\forall X) E_n = E'_n$$

$$(\forall X) \{(l_i = E_i)_{i=1,n-1}, l_n = E_n\} = \{(l_i = E_i)_{i=1,n-1}, l_n = E'_n\}$$

375
Exercise 251. Using the rules for typing and for equation derivation above, show formally that the expression

\[
(\lambda \ p : \{\text{age} : \text{nat}\} \ . \ (p.\text{age}))
\]

\[
\{\text{name} = \text{‘John Smith’}, \text{age} = 27, \text{height} = 180\ldots\}
\]

types to nat and is equal to 27.

Proposition 31. *(Type preservation.*) If \((\forall X) \ E = E'\) is derivable and \(X \triangleright E : t\) then \(X \triangleright E' : t\).

As an operational consequence of the property above, we obtain the so-called “subject reduction” property for the transition relation \(\rightarrow\) obtained by orienting the equations:

Corollary 16. If \(X \triangleright E : t\) and \(E \rightarrow E'\) then \(X \triangleright E' : t\).

Subtyping and Other Language Features

Subtyping occurs many places in computer science. Therefore, it is important to understand how subtyping interacts with different other features that also appear often in practice.

We next investigate the relationship between subtyping and several other interesting and important programming language features, including built-in types, lists, references, arrays, type casting and a bit with objects. The list of features can continue. In fact, it is customary that programming language designers analyze the effectiveness of new conceptual developments by studying their interaction with subtyping.

Built-in Types

Basic subtyping may be given by many other features besides records. As you know, programming languages usually provide built-in, also called basic, types, such as bool, int, real, etc.

Programmers find it very convenient to assume subtyping on some basic types, e.g., bool \(\leq\) int. In fact, in many languages, the constant \texttt{true} is represented as integer 1 and \texttt{false} as 0. This way, one can use a bool expression whenever an integer expression is expected. For example, with a Boolean variable \(b\), one can write an expression: \(scale \ast b\), which evaluating to wither 0 or \(scale\), depending on whether \(b\) is false or true. To support this feature, we need that bool \(\leq\) int. Other common subtype relationships are nat \(\leq\) int, or int \(\leq\) real.

Lists

We can also introduce list types into our type system:

\[
\text{Type ::= \ldots | list Type}
\]

The subtyping rule for lists is

\[
\frac{t_1 \leq t_2}{\text{list } t_1 \leq \text{list } t_2}
\]

Although this rule seems straightforward and intuitive, it only works when we build and use lists without applying any change on them. As we will next see in the discussion of references and arrays, some restrictions are raised by changing values at given locations in the list.

References

Most languages allow assignments of new values to existing, i.e., already declared, names. In many languages this is supported by introducing the notion of reference (also called location or cell index). We next extend our simply typed \(\lambda\)-calculus to support references and assignments:

\[
\text{Type ::= \ldots | Ref Type | Unit}
\]

\[
\text{Exp ::= \ldots | ref Exp | !Exp | Exp := Exp}
\]
Therefore, we allow explicit types for references. For example, \( \text{Ref} \ (\text{nat} \to \text{nat}) \) is a type for references to, or locations storing, functions \( \text{nat} \to \text{nat} \). Values of reference types are just like any other values in the languages, that is, they can be passed to and returned as results of functions.

Together with references and assignments, side effects are unavoidable. \( \text{Unit} \) is the type of expressions, such as assignments, that are intended to be used just for their side effects; these are not supposed to evaluate to any particular value. In other words, one may read \( \text{Unit} \) as “no type”.

Three new expression constructors are introduced.

\( \text{ref} \ E \) evaluates to a new location, say \( L \), where the expression \( E \) is also stored; this is equivalent to the “new” construct in object oriented languages, or to “malloc” in C. Depending upon the particular evaluation strategy desired in one’s language, the expression \( E \) stored at \( L \) may be already evaluated to some value. The language construct \( \text{ref} \) is also called referencing.

\( ! E \), which expects \( E \) to evaluate to a location, say \( L \), returns the expression/value stored at \( L \). The language construct \( ! \) is also called dereferencing.

\( E := F \) first evaluates \( E \) to a location, say \( L \), and then stores \( F \) at location \( L \). Again, depending upon the desired evaluation strategy, one may first evaluate \( F \) to some value \( V \) and then store \( V \) at \( L \).

To give a full equational semantics of our \( \lambda \)-calculus language extended with references, we would need to follow the same approach as in FUN, that is, to introduce infrastructure for stores, to evaluate \( \lambda \)-abstractions to closures, etc. We do not do this here, because it would essentially duplicate what we’ve already done in the definition of FUN. Instead, we just focus on aspects related to typing in the context of references.

**Typing references.**

The typing rules for references are straightforward:

\[
\begin{align*}
X \vdash E : t \\
X \vdash \text{ref} \ E : \text{Ref} \ t
\end{align*}
\]

\[
\begin{align*}
X \vdash E : \text{Ref} \ t \\
X \vdash !E : t
\end{align*}
\]

\[
\begin{align*}
X \vdash E : \text{Ref} \ t \quad X \vdash E' : t \\
X \vdash E := E' : \text{Unit}
\end{align*}
\]

The subtleties of typing in the context of references come from their interaction with subtyping.

**Subtyping and references.**

The question to be asked here is how, and under what conditions, can one derive subtyping relations of the form \( \text{Ref} \ t_1 \leq \text{Ref} \ t_2 \)? In other words, when can a reference to an expression/value of type \( t_1 \) be safely used in a context where a reference to an expression/value of type \( t_2 \) is expected?

There are two concerns regarding the use of a reference expression/value \( R \) of type \( \text{Ref} \ t_1 \) when a reference expression/value of type \( \text{Ref} \ t_2 \) is expected:

1. If \( R \) is dereferenced (read) in a context, such as, for example, in \( 3+!R \), the expression/value stored at \( R \) should be safely usable in that context, where an expression/value of type \( t_2 \) is expected; therefore, \( t_1 \leq t_2 \).

2. If \( R \) is assigned (written) in a context, for example using an assignment of the form \( R := E \) with \( E \) of type \( t_2 \), then since there can be other places in the program expecting (by dereferencing \( R \)) the expression/value at location \( R \) to have the declared type \( t_1 \), one deduces that \( t_2 \leq t_1 \).
Therefore, the only safe possibility to have $Ref\ t_1 \leq Ref\ t_2$ is that $t_1 = t_2$, which obviously implies $Ref\ t_1 = Ref\ t_2$. We conclude from here that reference types admit no proper subtypes.

While this is an elegant conclusion in what regards the implementation of a type system, because one basically needs to do absolutely nothing to support it, it is very important to understand the deep motivations underlying it.

Don’t speak unless you can improve on the silence

Spanish Proverb

While this may seem rather straightforward, when references and types live together in a language almost nothing is simple enough to avoid misunderstandings or subtle problems.

Arrays

Arrays can be very easily added to a language, at least in what regards their typing. For example, in our language we can extend both the types and the expressions as follows:

\[
\text{Type} ::= \ldots \mid \text{Array} \text{Type}
\]

\[
\text{Exp} ::= \ldots \mid \text{Exp[Exp]} \mid \text{Exp[Exp]} := \text{Exp}
\]

Essentially, arrays are similar to references. In fact, in many languages, arrays are nothing but references pointing to the first location of a block of memory. Consequently, by an analysis of subtyping similar to that for references, we can infer that the only reasonable way to have $\text{Array} t_1 \leq \text{Array} t_2$ is that $t_1 = t_2$, meaning that array types have no proper subtypes. This conclusion can similarly be applied to lists when one is allowed to write at specific locations in lists (with statements of the form $\text{car}\ E := E'$, $\text{car}(\text{cdr}\ E) := E'$, etc.).

Some programming language designers, while still adopting type systems for their languages, find some of the above (static) subtyping restrictions too strong, arguing that they limit the use of references, arrays, lists, etc. In fact, designers of programming languages tend to easily become “religious”; for example, those designing untyped or dynamically typed languages think of static typing as a serious impediment the programmers have to deal with in one way or another.

Some languages split the task of type checking into a static component and a dynamic one. For example, Java only takes the deferencing (reading) into account during its static type checking, and checks every write at runtime to maintain the type safety. But this is also considered by some researchers as a design flaw…

Type Casting

Type casting allows one to assign to terms types that type checkers may not be able to find statically. One can regard casting as a type annotation which helps the type checker analyze the program. At some extent, unless a language admits some form of dynamic type checking that can lead to different executions when expressions are evaluated to values of different types (such as instanceof checks), one can regard all the type declarations as just annotations to help a particular static program analysis tool, the type checker, analyze the program.

The corresponding syntax for type casting in our language can be:

\[
\text{Exp} ::= \ldots \mid (\text{Type}) \text{Exp}
\]

We do not discuss the formal (equational) semantics of casting here, but, intuitively, $(t)E$ is simply equal to $E$ when $E$ can be shown of type $t$. In some cases one may be able to show statically that $E$ has type $t$ or that $E$ cannot have type $t$, in which case the type casting may be simply dropped or a static error reported, but
in general the problem is undecidable. In practice, a dynamic checker, or a monitor, is inserted to ensure that the type of \( E \) is indeed as claimed; if not, an error or an exception is generated. The benefit of this dynamic check is that the static type checker can then assume that \( E \) has indeed the claimed type \( t \) and can therefore continue unperturbed the type checking process on the rest of the program. The typing rule of casting is then simply as follows:

\[
\frac{X \triangleright E : t_2}{X \triangleright \langle t_1 \rangle E : t_1}
\]

Therefore, as far as an expression type checks to any type, it can also be cast to any other type, but a dynamic check still needs to be performed. For example, \( \langle \{\text{age} : \text{nat}\} \rangle x.\text{age} \) is assumed of type \( \text{nat} \) for static type checking purposes, but it may cause a runtime error if the type \( t \) of \( x \) is not a record type containing a field \( \text{age} \) of type \( \text{nat} \).

Exercise 252. Give a simple program containing the expression \( \langle \{\text{age} : \text{nat}\} \rangle x.\text{age} \) which types and executes correctly, but which would not type if one replaced the expression above by just \( x.\text{age} \).

To facilitate casting, many languages have a top type, like the \textit{Object} class in Java.

Syntax-Directed Subtyping

The subtyping rules, together with the typing rules in the context of subtypes, gave us a logical means to entail type judgements of the form \( X \triangleright E : t \). However, note that the type system is bound to the fact that all three components of the type judgement, namely \( X \), \( E \), and \( t \), need to be available.

For some typed frameworks, for example simply typed \( \lambda \)-calculus, as we know it is relatively trivial to translate the typing rules into a typing algorithm, calculating for a given expression \( E \) in a given type environment \( X \) a type \( t \), if the expression is indeed well-typed, and reporting an error otherwise.

In the context of subtyping, one needs some additional work to obtain an effective typing algorithm.

Let us first understand what are the complications that appear when one tries to type a program within a language with subtyping.

Unlike typing in the context of our previous, non-subtyped languages, in the context of subtypes the application of the typing rules is \textit{not} deterministic, i.e., \textit{not} syntax-directed. Consider for example the \( \lambda \)-expression, say \( \text{mkYounger} \),

\[
\lambda x : \{\text{age} : \text{nat}, \text{height} : \text{nat}\}.
\{\text{age} = x.\text{age} - 20, \text{height} = x.\text{height} + 3\}
\]

and suppose that we would like to derive that it can have the type \( \{\text{age} : \text{nat}, \text{height} : \text{nat}\} \to \{\text{age} : \text{nat}\} \), say \( t \). There are two different ways to do it:

1) One can first apply typing rules for as long as possible, and then “ask for the help” the subtyping rules:

\[
\begin{align*}
(1) & \quad \emptyset \triangleright \text{mkYounger} : \{\text{age} : \text{nat}, \text{height} : \text{nat}\} \to \{\text{age} : \text{nat}, \text{height} : \text{nat}\} \\
(2) & \quad \{\text{age} : \text{nat}, \text{height} : \text{nat}\} \to \{\text{age} : \text{nat}, \text{height} : \text{nat}\} \leq \{\text{age} : \text{nat}, \text{height} : \text{nat}\} \to \{\text{age} : \text{nat}\} \\
(3) & \quad \emptyset \triangleright \text{mkYounger} : t
\end{align*}
\]

2) Alternatively, a subtyping rule can be applied earlier, before typing the outermost expression:
In order for the entailment system to immediately provide a typing algorithm, one would like to have a well-defined, deterministic way to apply the typing rules by just examining the syntax of the program without any search. This is also called syntax-directed typing.

Of course, one may argue that, in the above example, choosing one of the two possible computations is not problematic, thanks to their confluence; hence one could pick, for instance, the lexicographically lower computation. But there are some rules in our context of subtyping that are really non-syntax-directed, in the sense that one has no clue where to find the catalyzers needed to continue computation.

Undeterministic subtyping rules.
Consider for instance the rules:

(≤-subtyping) \[ \frac{t_1 \leq t \quad t \leq t_2}{t_1 \leq t_2} \] How to pick a \( t \) when applying this rule?

(subsumption) \[ \frac{X \triangleright E : t_1 \quad t_1 \leq t_2}{X \triangleright E : t_2} \] How to pick a \( t_1 \)?
Structure/syntax of \( E \) is not taken into account.

Can one change the rules so that typing becomes syntax-directed and mechanical? The answer is yes.

Eliminating the bad rules.
A first interesting observation is that the (≤-transitivity) rule is not needed. The intuitive reason underlying this claim is that the subtyping relation derived using the other subtyping rules, excluding (≤-transitivity), is already transitive.

Exercise 253. Prove formally the claim above.

A second important observation is that we can also eliminate the other problematic rule, (subsumption), by carefully inspecting its usage. This rule really only needs to be used to type check function applications: \((λx : t. E)E'\) requires that the type of \(E'\) is a subtype of \(t\). Then one can simply eliminate the problematic (subsumption) rule and instead modify the \(λ\)-abstraction application rule into the following rule. To reflect the fact that the newly obtained type system is different from the original one, we use a different but closely related syntax, namely \(X \triangleright E : t\), to denote the type judgements derived with the latter type system:

(≤-application) \[ \frac{X \triangleright E_1 : t_1 \rightarrow t_1', \quad X \triangleright E_2 : t_2, \quad t_2 \leq t_1}{X \triangleright E_1E_2 : t_1'} \]

Note that the (≤-application) rule above is syntax-driven, in the sense that the syntax of the goal type judgment (an application), tells us precisely what to do next: calculate the type of the two expressions involved and then derive the corresponding subtype relation.

Proposition 32. Prove that the resulting typing system above has the following properties:

- For each expression \( E \) and type assignment \( X \), one can derive \( X \triangleright E : t \) for a at most one type \( t \);
• **(Soundness)** \( X \vdash E : t \Rightarrow X \triangleright E : t; \)

• **(Completeness)** \( X \triangleright E : t \Rightarrow X \vdash E : t', \) where \( t' \leq t. \)

Therefore, in terms of the original type system, the new type system derives the most concrete type of an expression.

**Syntax-directed subtyping algorithm.**

The important proposition above immediately provides an algorithm to decide whether, under a type assignment \( X, \) an expression \( E \) has a type \( t: \)

First derive \( X \vdash E : t' \) and then check whether \( t' \leq t; \) if any of the two fails than \( E \) cannot have the type \( t \) in the type environment \( X. \)

**Exercise 254.** Define the syntax-directed typing algorithm above in Maude. You need to define both the subtyping relation and the typing rules equationally. Your definition should be executable.

**Typing Conditionals**

In simply typed \( \lambda \)-calculus we had to define one conditional constant \( \text{cond}_t : t \rightarrow t \rightarrow \text{bool} \) for each type \( t. \) With subtyping, like with universal polymorphism, we only need to define one generic conditional. Precisely, we can add a \( \text{Top} \) constant type defined to be the most general type together with a subtyping rule

\[
\begin{array}{c}
\text{(top)} \\
\hline
\end{array}
\]

\( t \leq \text{Top} \)

and then define just one constant conditional expression, \( \text{cond} : \text{bool} \rightarrow \text{Top} \rightarrow \text{Top} \rightarrow \text{Top}. \) (This actually follows a general technique to simulate universal polymorphism with subtyping).

Previously, the rule for typing conditionals was the following:

\[
\frac{X \triangleright C : \text{bool}, \ X \triangleright E_1 : t, \ X \triangleright E_2 : t}{X \triangleright \text{cond} \ C \ E_1 \ E_2 : t}
\]

This rule still works in the context of subtyping, but note, however, that one is expected to use subsumption to lift the possibly more concrete types of \( E_1 \) and \( E_2 \) to some common supertype. In fact, the conditional expression can be derived any type that is a supertype of the most concrete types of \( E_1 \) and \( E_2. \) For example, consider the following expression (“keep one’s height if one is older than 10, otherwise keep one’s weight”):

\[
\text{cond} \ (x. \ age > 10) \\

\{ \text{name} = x.\text{name}, \ \text{age} = x.\text{age}, \ \text{height} = x.\text{height} \} \quad \{ \text{name} = x.\text{name}, \ \text{age} = x.\text{age}, \ \text{weight} = x.\text{weight} \}
\]

One cannot apply directly the typing rule of the conditional, because the two branches have different types. But by subsumption one can first calculate some common type to both branches, such as \{\text{name} : \text{String}, \ \text{age} : \text{nat} \}, or \{\text{name} : \text{String} \}, or \{\text{age} : \text{nat} \}, or even \{\}, and then apply the typing rule for conditional.

The limitations of the typing rule for conditional above becomes clear in the context of syntax-directed typing, where one calculates the most concrete types of the two branches and then one wants to calculate a type for the conditional.

First, note that under syntax-directed typing, the typing rule for conditional above is close to useless, because the most concrete types of the branches may be different.

Second, what should the type of the conditional actually be, knowing the types \( t_1 \) and \( t_2 \) of its branches? Since we want to calculate the type of the conditional statically, unless using sophisticated theorem provers
which typically do not to scale), we cannot know which of the two branches would be taken during an actual execution.

One possibility would be to consider both branches separately and ensure that the program would type regardless of which branch is taken. Unfortunately, the number of possibilities to analyze doubles with each conditional in the program. Therefore, despite its precision in analysis, this exhaustive approach would hardly have any practical use.

An extreme possibility would be to say that the type of the conditional is \textit{Top}, because we do not know which of the two branches is taken. The problem with this aggressive approach is that all the type information about the two branches is lost, so one may need casting to explicitly concretize the type of the conditional to the expectations of a particular context.

The practical solution here is to accept losing some precision but not all of it. Any common supertype of \( t_1 \) and \( t_2 \) is clearly better than \textit{Top}. This suggests that we should actually pick the \textit{least common supertype} of \( t_1 \) and \( t_2 \), written \( t_1 \lor t_2 \), as the type of the conditional; since the subtype relation is a partial order, the least common supertype of \( t_1 \) and \( t_2 \), also called their \textit{join} type, is nothing but the least upper bound of \( t_1 \) and \( t_2 \) with respect to the subtyping relation. With this, the syntax-driven typing rule of the conditional is the following:

\[
\frac{X \vdash C : \text{bool} \quad X \vdash E_1 : t_1 \quad X \vdash E_2 : t_2}{X \vdash \text{cond} \ C \ E_1 \ E_2 : t_1 \lor t_2}
\]

Thus our sample conditional expression discussed above types to \{\textit{name} : \textit{String}, \textit{age} : \textit{nat}\}.

\textbf{Exercise 255}. (continuation of Exercise 254) Define (also in Maude) the join operation on types and add the universal conditional to the language together with its syntax-driven typing.

\section*{Subtypes and Objects}

Objects and the object-oriented (OO) paradigm form an important player in today’s theory and practice of programming languages. Objects are considered by many software engineers crucial in the process of software development, because they improve the modularity and reusability of code.

Essentially, an object \textit{encapsulates} a state and provides the outside world an \textit{interface} to partially manipulate its state, like for example to access or modify parts of it. We next show how objects and some OO concepts can be supported, without any additional machinery, by typed \( \lambda \)-calculus with subtyping and references.

That should not mean that we are actually claiming that typed high-order languages should replace OO languages. Similarly, the fact that all programming language paradigms can be supported by rewriting does not mean that rewriting can replace all these.

\textbf{Syntax}

The syntax is left almost unchanged:

\[
\begin{align*}
\text{Type} & ::= \ldots | \text{Ref Type} | \text{Unit} \\
\text{Exp} & ::= \ldots | \text{ref Exp} | \text{Exp} | \text{Exp} \vdash \text{Exp} | ()
\end{align*}
\]

The only increment here is () that is used for calling functions without parameters. This feature is not required, since we can pass a dummy argument that is ignored in the function, but it is more natural and concise for programming. The type of () is \textit{Unit}, as expected:
Representing objects

Let us consider a very simple example, that of a counter object which contains a private integer value and provides two functions to the outside world, namely get() to get the value of the integer and inc() to increment it. Once such a counter object is created, say c, one would like to be be able to write expressions like c.get() and c.inc(), with the obvious meaning.

In our framework, it is then natural to represent objects as records, containing a field for each operation allowed to be visible, or part of the interface, to the outside world. In our particular case, an already created counter object can be represented as a record \( \{ \text{get} = \ldots, \text{inc} = \ldots \} \). The concrete integer value is not intended to be accessed directly from the outside, so it does not have a corresponding field in the record; nevertheless, one still wants to access it indirectly, using the field get.

For the time being, let us assume that the state of an object has already been created. In our case of the simple counter, since the state consists of just one integer, let us assume that it has already been created in an outer scope and that a reference \( x \) to it is available. Then the representation of our already created object is:

\[
\{ \text{get} = \lambda : \text{Unit} \cdot x, \text{inc} = \lambda : \text{Unit} \cdot x := (!x + 1) \}
\]

We will later see that the state of an object can be actually kept in a record (of references), which is made visible to the functions. As we already know by now, the evaluation strategies play a much more important role in the design of a language in the presence of side effects. Since OO programming is all about side effects, we impose a call-by-value evaluation strategy in our language. Also, function bodies are evaluated only when functions are called; otherwise, the body of the function inc can be evaluated indefinitely and therefore the counter incremented in an uncontrolled manner.

Creating objects

One needs to be able to create an object before one can use it. Intuitively, creation of an object comes to initializing the state of the object. Considering again our example, we need to create an integer reference with some value and give that reference to the object. To achieve this, we can use a macro, say createCounter, defined as follows:

\[
(\lambda x : \text{Ref nat} . \\
\{ \text{get} = \lambda : \text{Unit} \cdot x, \text{inc} = \lambda : \text{Unit} \cdot x := \text{succ}(!x) \})
\]

Because of the assumed call-by-value evaluation strategy, each time the macro above is evaluated the following happen (recall also the definition of FUN): (1) a location is created and the integer value is stored at that location; (2) that location is bound to \( x \) in an environment, say \( \text{Env} \), in which the record is evaluated; (3) the two functions in the record are evaluated to corresponding closures, each freezing the environment \( \text{Env} \) (further calls of these functions will therefore see the same location of \( x \)).

In short, this macro creates a counter object with the initial value of 1, which types as expected:

\[
\emptyset \triangleright \text{createCounter} : \{ \text{get} : \text{Unit} \to \text{nat}, \text{inc} : \text{Unit} \to \text{Unit} \}
\]

One would like to have the possibility to create many objects with a similar functionality, in our case many counters, without having to write the creation code over and over again. In a similar fashion to the new construct in object oriented programming, one can define a newCounter macro as the \( \lambda \)-abstraction \( \lambda x : \text{Unit} \cdot \text{createCounter} \). One can now create counter objects in any context, by simply invoking newCounter(). Note that a new state is indeed created with each invocation of newCounter.
Exercise 256. Define another macro for creating counter objects, namely one which takes as argument an integer and creates a counter object initialized to that integer. Therefore, newCounter(7) should create a counter object initialized to 7.

Subtyping objects
We know from OO that objects that are instances of subclasses can also be regarded as instances of superclasses. Without yet discussing classes and sub/superclasses, let us first see how an object having more information than another one, is actually typed to a subtype of the later object in our framework.

Consider an enriched counter, which, besides the usual get and inc methods, has a method reset which resets the counter to 0. The new enriched counter objects have therefore the type:

\[ \{ \text{get : Unit} \to \text{nat}, \ \text{inc : Unit} \to \text{Unit}, \ \text{reset : Unit} \to \text{Unit} \} \]

Therefore, the type of the enriched counters is a subtype of the simpler counters, which is consistent with the OO intuitions for objects and their types/classes. One can easily write a \( \lambda \)-abstraction for creating reset counter objects, say newResetCounter:

\[
(\lambda \_ : \text{Unit} . \\
(\lambda x : \text{Ref nat} . \\
\{ \text{get} = \lambda \_ : \text{Unit} . !x, \\
\text{inc} = \lambda \_ : \text{Unit} . x := (!x + 1), \\
\text{reset} = \lambda \_ : \text{Unit} . x := 0 \}) (\text{ref 1})
)
\]

Let us next understand how classes can be encoded in our typed high-order framework.

Classes
In OO programming languages, classes typically are regarded as the types of objects, while objects are regarded as instances of classes. As it turns out, these words can mean almost everything, depending upon context, who says them and who listens to them.

It is customary though to consider that classes contain a certain kind of functionality and structure, while an object of a class contains, or encapsulates, a concrete state together with a handle to the class that it is an instances of. By a state, one typically means a type-consistent assignment of values to fields.

With this model in mind, we can refine our definition of an object as follows.

1. First introduce a record type containing all the intended fields of a class; since the values of these fields may be changed during the life-time of an object, the types of the fields must be reference types. In the case of our counter we have only one field, so let us introduce the record type \( \{ x : \text{Ref nat} \} \) and call it CounterRef. Whenever an object is created as an instance of a class, one should first create a record allocating concrete values to the fields;

2. Next define the class itself as a \( \lambda \)-abstraction taking a value record type as above and adding methods to it to taste. In our case, we can define a class CounterClass as follows:

\[
\lambda r : \text{CounterRef} . \\
\{ \text{get} = \lambda \_ : \text{Unit} . (r.x), \\
\text{inc} = \lambda \_ : \text{Unit} . (r.x) := (r.x + 1) \}
\]
Classes therefore evaluate to closures and type to function types taking field record types to method record type. For example, the class `CounterClass` types to

\[\text{CounterRef} \rightarrow \{\text{get} : \text{Unit} \rightarrow \text{nat}, \quad \text{inc} : \text{Unit} \rightarrow \text{Unit}\}\]

3. Create objects by passing desired field records to classes. In our case, a counter with integer value 1 is created by simply evaluating `Counter(x = ref 1)`. Also, one can define `newCounter` as `\(\lambda \_ : \text{Unit}. \text{Counter}(x = \text{ref 1})\)`, and so on.

This way, we therefore have a means to devise classes and then to create objects as instances of them. The type system of our calculus maintains a certain discipline in how objects are created and used, but it may still allow one to develop programs which one would rather want to reject.

**Subclasses/Inheritance**

*Inheritance*, or the process of extending existing classes with functionality, is an important, if not the most important, feature of the OO paradigm. The extended classes, despite the fact that they add functionality to the extended classes, are actually called *subclasses*. This terminology is consistent with our terminology for subtypes. Let us next see, again via an example, how we can simulate inheritance within our framework in a consistent way, in the sense that the type system will rank subclasses as subtypes.

Let us define a class `ResetCounterClass` which extends `CounterClass` by adding a reset method:

\[\lambda r : \text{CounterRef}. \quad (\lambda \text{super} : \_ \_ . \quad \{\text{get} = \text{super}.\text{get}, \quad \text{inc} = \text{super}.\text{inc}, \quad \text{reset} = \lambda _ : \text{Unit}. (r.x = 0)\})\]

\((\text{CounterClass} r)\)

**Exercise 257.** What is the type of `super` above (the red question mark)? Type the `ResetCounterClass` above and show that it is a subtype of `CounterClass`.

The use of `super` above is not required, but it helps to highlight the relationship between the subclass and the superclass.

However, although we would like to see a one-to-one relationship between the notions of subclass and corresponding subtype in our framework, the simulation of the OO paradigm above is so flexible that it actually allows one to also remove methods from the superclass, thus breaking the subtyping relationship, disallowing the use of the subclass where the superclass is expected, etc. To prevent that, we may need to provide more powerful mechanisms or primitives in the language, e.g., extends/inherits, together with appropriate typing rules/policies.

**Exercise 258.** Comment on the encoding of OO programming in high-order typed languages presented above. Tell straight your opinion and think of desirable OO features that could not be supported this way. Also, show how self-reference can also be supported by allowing recursion (\(\mu\)) in the language. How about dynamic versus static method dispatch?
Bibliography


387


1982; and Report CSLI-84-15, Center for the Study of Language and Information, Stanford University, September 1984.


390


discuss defining complete languages versus constructs, separated in modules

new approaches versus traditional approaches; after all, RL is only 10 years younger than SOS . . .

model-based semantics in addition to an operational one; discuss the difference, because apparently the initial model is the transition system advocated by SOS . . .

additional notation brings additional benefits; one is not forced to use, but one may find advantages in using it

sometimes an order of application of rules may be desired; add strategy language?
Appendix A

Maude Stuff

A.1 Defining “Builtin” Modules for Maude Language Definitions

When defining language semantics in Maude, for example using the operational semantics styles discussed in Chapter 3, one may naturally want to use the existing Maude data-types and libraries, particularly because they are well-engineered and thus can lead to efficient interpreters and tools for the defined languages. However, one needs to exercise care when importing Maude builtin modules, because it is very difficult, if not impossible, to reuse the names of the builtin operations as programming language constructs. The reason is that there is a very high chance that one wants to give those language constructs different attributes from those of the builtin operations, but Maude, for good reasons that go beyond our purpose in this book, does not allow it. For example, one may want to include an addition operation \( \_+\_ \) in one’s language, but one may not want it to be associative, commutative and have \( 0 \) as identity, like the \( \_+\_ \) operation provided by the builtin Maude module INT.

In this section we give some Maude 2.4 specific hints on how one can appropriately redefine the Maude builtin modules to work smoothly with the Maude language semantics in Chapter 3. We advise the reader to run Maude without including its prelude, that is, with the command:

\[
\text{maude -no-prelude <language-semantics>.maude}
\]

This way, no warnings will be issued when we redefine the builtin modules INT and BOOL. If for some reason one really needs to include the Maude prelude at start, then one should at least make sure that the BOOL module is not automatically included in other modules using the Maude command:

FIXTHIS

Figure A.1: The Maude builtin modules used in the IMP semantics in Chapter 3

395
We can now define our own and safe BOOL module as follows:

```plaintext
mod BOOL is
  sort Bool .
  op true : -> Bool [ctor special (id-hook SystemTrue)] .
  op false : -> Bool [ctor special (id-hook SystemFalse)] .
  op if_then_else_fi : Bool Universal Universal Universal -> Universal [poly (2 3 0)
    special (id-hook BranchSymbol term-hook 1 (true) term-hook 2 (false))].
  op _==_ : Universal Universal -> Bool [poly (1 2) prec 51 special (
    id-hook EqualitySymbol term-hook equalTerm (true) term-hook notEqualTerm (false))].
  op _=/=_ : Universal Universal -> Bool [poly (1 2) prec 51 special (
    id-hook EqualitySymbol term-hook equalTerm (false) term-hook notEqualTerm (true))].
endm
```

The module above includes only a few components from the actual Maude builtin BOOL module. More precisely, it discards all the specific Boolean operators for conjunction, negation, implication, etc., because one may want to define these as language constructs in one's language, possibly having a different semantics. For example, the conjunction _and_ of IMP in Chapter 3 is short-circuited, so in particular it is not commutative (while the Maude BOOL builtin _and_ is commutative). If one really needs to include the original Maude BOOL builtin operations, then we recommend that one change their names, like we do within the next module.

Like the module BOOL above, the module INT below defines only a portion of the Maude INT builtin module of integers. Unlike in BOOL, this time we also include some library operations on integers as part of the new builtin INT, but, however, we change their names by appending the word “Int” to their original names, e.g., _+Int_, _<=Int_, etc.:

```plaintext
mod INT is including BOOL .
  sort Int .
  op 0 : -> Int [ctor].
  op s_ : Int -> Int [iter ctor special (id-hook SuccSymbol term-hook zeroTerm (0))].
  op _-Int_ : Int -> Int [ctor special (id-hook MinusSymbol op-hook succSymbol (s_ : Int ˜> Int) op-hook minusSymbol (-Int_ : Int ˜> Int))].
  op _+Int_ : Int Int -> Int [assoc comm prec 33 special (id-hook ACU_NumberOpSymbol (+) op-hook succSymbol (s_ : Int ˜> Int) op-hook minusSymbol (-Int_ : Int ˜> Int))].
  op _/Int_ : Int Int -> Int [prec 31 gather (E e) special (id-hook NumberOpSymbol (quo) op-hook succSymbol (s_ : Int ˜> Int))].
  op _<=Int_ : Int Int -> Bool [prec 31 gather (E e) special (id-hook NumberOpSymbol (quo) op-hook succSymbol (s_ : Int ˜> Int))].
endm
```

This way, one can still have full access to Maude's builtin libraries, but one elegantly avoids name clashes with homonymous operations that one may want to define as part of one's language syntax.

Finally, we define the VAR module. As discussed in Section [3.1.1], we want it to provide a sort Id and at least each letter in the alphabet as a constant of sort Id to be used in writing readable programs. We additionally include all the Maude builtin quoted identifiers (e.g., 'abc, 'abc123, etc.) as variables, so that one can use them as well:

```plaintext
mod VAR is
  sort Id .
  ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Id [format (g o)].
  op <Qids> : -> Id [special (id-hook QuotedIdentifierSymbol)] .
endm
```

396
Note that the module above also includes the Maude definition of lists of variables; these may be needed for variable declarations, as we did in Section 3.1.1 for IMP.