A rewriting logic approach to operational semantics

Traian Florin Șerbănuță, Grigore Roșu, José Meseguer

Abstract

This paper shows how rewriting logic semantics (RLS) can be used as a computational logic framework for operational semantic definitions of programming languages. Several operational semantics styles are addressed: big-step and small-step structural operational semantics (SOS), modular SOS, reduction semantics with evaluation contexts, continuation-based semantics, and the chemical abstract machine. Each of these language definitional styles can be faithfully captured as an RLS 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 RLS theory. A major goal of this paper is to show that RLS does not force or pre-impose any given language definitional style, and that its flexibility and ease of use makes RLS an appealing framework for exploring new definitional styles.

1. Introduction

This paper is part of the rewriting logic semantics (RLS) project (see [57,56] and the references there in). The broad goal of the project is to develop a tool-supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on rewriting logic [50].

Any logical framework worth its salt should be evaluated in terms of its expressiveness and flexibility. Regarding expressiveness, a very pertinent question is: how does RLS express various approaches to operational semantics? In particular, how well can it express various approaches in the SOS tradition? The goal of this paper is to provide an answer to these questions. Partial answers, giving detailed comparisons with specific approaches have appeared elsewhere. For example, [48,91] provide comparisons with standard SOS [70]; [55] compares RLS with both standard SOS and Mosses’ modular structural operational semantics (MSOS) [65]; and [50] compares RLS with chemical abstract machine (Cham) semantics [8]. However, no comprehensive comparison encompassing most approaches in the SOS tradition has been given to date. To make our ideas more concrete, in this paper we use a simple programming language, show how it is expressed in each different definitional style, and how that style can be faithfully captured as a rewrite theory in the RLS framework. We furthermore prove correctness theorems showing the faithfulness of the RLS representation for each style. Even though we exemplify the techniques and proofs with a simple language for concreteness’ sake, the process of representing each definitional style in RLS and proving the faithfulness of the representation is completely general and mechanical, and in some cases like MSOS

© 2009 Published by Elsevier Inc.
has already been automated [19]. The range of styles covered includes: big-step (or natural) SOS semantics; small-step SOS semantics; MSOS semantics; context-sensitive reduction semantics; continuation-based semantics; and Cham semantics.

Concerning flexibility, we show that each language definitional style can be used as a particular definitional methodology within rewriting logic. It is not our point in this paper to argue whether certain rewriting-logic specific methodologies are in some ways better or worse than others, but simply to enable the language designer to use his/her favorite techniques within rewriting logic with the benefit of a unified logic and generic tool support. Other than that, representing a language definitional style in rewriting 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 rewriting logic inherits the same benefits and limitations that the original definitional style had.

1.1. Challenges

Any logical framework for operational semantics of programming languages has to meet strong challenges. We list below some of the challenges that we think any such framework must meet to be successful. We do so in the form of questions from a skeptical language designer, following each question by our answer on how the RLS framework meets each challenge question. The full justification of many of our answers will become clearer in the body of the paper.

(1) Q: Can you handle standard SOS?
   A: As illustrated in Sections 5 and 6 for our example language, and also shown in [48,91,55] using somewhat different representations, both big-step and small-step SOS definitions can be expressed as rewrite theories in RLS. Furthermore, as illustrated in Section 7 for our language, and systematically explained in [55], MSOS definitions can also be faithfully captured in RLS.

(2) Q: Can you handle context-sensitive reduction?
   A: There are two different questions implicit in the above question: (i) how are approaches to reduction semantics based on evaluation contexts [e.g., [96]] represented as rewrite theories? and (ii) how does RLS support context-sensitive rewriting in general? We answer subquestion (i) in Section 8, where we illustrate with our example language a general method to handle evaluation contexts in RLS. Regarding subquestion (ii), it is worth pointing out that, unlike standard SOS, because of its congruence rule, rewriting logic is context-sensitive and, furthermore, using frozen operator arguments, reduction can be blocked on selected arguments (see Section 2). Rewriting logic provides no support for matching the context in which a rewrite rule applies and to modify that context at will, which is one of the major strengths of reduction semantics with evaluation contexts. If that is what one wants to do, then one should use the technique in Section 8 instead.

(3) Q: Can you handle higher-order syntax?
   A: Rewriting logic, cannot directly handle higher-order syntax with bindings and reasoning modulo $\alpha$-conversion. However, it is well known that higher-order syntax admits first-order representations, such as explicit substitution calculi and de Bruijn numbers, e.g., [1,7,82]. However, the granularity of computations is changed in these representations; for example, a single $\beta$-reduction step now requires additional rewrites to perform substitutions. In rewriting logic, because computation steps happen in equivalence classes modulo equations, the granularity of computation remains the same, because all explicit substitution steps are equational. Furthermore, using explicit substitution calculi such as CINNI [82], all this can be done automatically, keeping the original higher-order syntax not only for $\lambda$-abstraction, but also for any other name-binding operators.

(4) Q: What about continuations?
   A: Continuations [34,71] are traditionally understood as higher-order functions. Using the above-mentioned explicit calculi they can be represented in a first-order way. In Section 9, we present an alternative view of continuations that is intrinsically first-order in the style of, e.g., Wand [95], and prove a theorem showing that, for our language, first-order continuation semantics and context-sensitive reduction semantics are equivalent as rewrite theories in RLS. We also emphasize that in a computational logical framework, continuations are not just a means of implementing a language, but can be used to actually define the semantics of a language.

(5) Q: Can you handle concurrency?
   A: One of the strongest points of rewriting logic is precisely that it is a logical framework for concurrency that can naturally express many different concurrency models and calculi [51,49]. Unlike standard SOS, which forces an interleaving semantics, true concurrency is directly supported. We illustrate this in Section 10, where we explain how Cham semantics is a particular style within RLS.

(6) Q: How expressive is the framework?
   A: RLS is truly a framework, which does not force on the user any particular definitional style. This is illustrated in this paper by showing how quite different definitional styles can be faithfully captured in RLS. Furthermore, as already mentioned, RLS can express a wide range of concurrent languages and calculi very naturally, without artificial encodings. Finally, real-time and probabilistic systems can likewise be naturally expressed [2,54,67].

(7) Q: Is anything lost in translation?
   A: This is a very important question, because the worth of a logical framework does not just depend on whether something can be represented “in principle”, but on how well it is represented. The key point is to have a very small repre-
sentational distance between what is represented and the representation. Turing machines have a huge representational distance and are not very useful for semantic definitions exactly for that reason. Typically, RLS representations have what we call “representational distance”, that is, what is represented and its representation differ at most in inessential details. In this paper, we show that all the RLS representations for the different definitional styles we consider have this feature. In particular, we show that the original computations are represented in a one-to-one fashion. Furthermore, the good features of each style are preserved. For example, the RLS representation of MSOS is as modular as MSOS itself.

Q: Is the framework purely operational?
A: Although RLS definitions are executable in a variety of systems supporting rewriting, rewriting logic itself is a complete logic with both a computational proof theory and a model-theoretic semantics. In particular, any rewrite theory has an initial model, which provides inductive reasoning principles to prove properties. What this means for RLS representations of programming languages is that they have both an operational rewriting semantics, and a mathematical model-theoretic semantics. For sequential languages, this model-theoretic semantics is an initial-algebra semantics. For concurrent languages, it is a truly concurrent initial-model semantics. In particular, this initial model has an associated Kripke structure in which temporal logic properties can be both interpreted and model-checked [53].

Q: What about performance?
A: RLS as such is a mathematical framework, not bound to any particular rewrite engine implementation. However, because of the existence of a range of high-performance systems supporting rewriting, RLS semantic definitions can directly be used as interpreters when executed in such systems. Performance will then depend on both the system chosen and the particular definitional style used. The RLS theory might need to be slightly adapted to fit the constraints of some of the systems. In Section 11, we present experimental performance results for the execution of mechanically generated interpreters from RLS definitions for our example language using various systems for the different styles considered. Generally speaking, these performance figures are very encouraging and show that good performance interpreters can be directly obtained from RLS semantic definitions. Although for this paper, we have used hand-made (though mechanical) translations to the presented implementation languages, we envision a toolkit which would use RLS as a common framework for different definitional styles having as back-ends multiple execution languages.

1.2. Benefits

Our skeptical language designer could still say,

So what? What do I need a logical framework for?

It may be appropriate to point out that he/she is indeed free to choose, or not choose, any framework. However, using RLS brings some intrinsic benefits that might, after all, not be unimportant to him/her.

Besides the benefits already mentioned in our answers to questions in Section 11, one obvious benefit is that, since rewriting logic is a computational logic, and there are state-of-the-art system implementations supporting it, there is no gap between an RLS operational semantics definition and an implementation. This is an obvious advantage over the typical situation in which one gives a semantics to a language on paper following one or more operational semantics styles, and then, to “execute” it, one implements an interpreter for the desired language following “in principle” its operational semantics, but using one’s favorite programming language and specific tricks and optimizations for the implementation. This creates a nontrivial gap between the formal operational semantics of the language and its implementation.

A second, related benefit, is the possibility of rapid prototyping of programming language designs. That is, since language definitions can be directly executed, the language designer can experiment with various new features of a language by just defining them, eliminating the overhead of having to implement them as well in order to try them out. As experimentally shown in Section 11, the resulting prototypes can have reasonable performance, sometimes faster than that of well-engineered interpreters.

A broader, third benefit, of which the above two are special cases, is the availability of generic tools for: (i) syntax; (ii) execution; and (iii) formal analysis. The advantages of generic execution tools have been emphasized above. Regarding (i), languages such as Asp+Sdf [87] and Maude [23] support user-definable syntax for RLS theories, which for language design has two benefits. First, it gives a prototype parser for the defined language essentially for free; and second, the language designer can use directly the concrete syntax of the desired language features, instead of the more common, but harder to read, abstract syntax tree (AST) representation. Regarding (iii), there is a wealth of theorem proving and model checking tools for rewriting/equational-based specifications, which can be used directly to prove properties about language definitions. The fact that these formal analysis tools are generic, should not fool one into thinking that they must be inefficient. For example, the LTL model checkers obtained for free in Maude from the RLS definitions of Java and the JVM compare favorably in performance with state-of-the-art Java model checkers [31,33].

A fourth benefit comes from the availability in RLS of what we call the “abstraction dial”, which can be used to reach a good balance between abstraction and computational observability in semantic definitions. The point is which computational granularity is appropriate. A small-step semantics opts for very fine-grained computations. But this is not necessarily the only or the best option for all purposes. The fact that an RLS theory’s axioms include both equations and rewrite rules provides the useful “abstraction dial”, because rewriting takes place modulo the equations. That is, computations performed
by equations are abstracted out and become invisible. This has many advantages, as explained in [57]. For example, for formal analysis it can provide a huge reduction in search space for model checking purposes, which is one of the reasons why the Java model checkers described in [31,33] perform so well. For language definition purposes, this again has many advantages. For example, in Sections 6 and 5, we use equations to define the semantic infrastructure (stores, etc.) of SOS definitions; in Section 8 equations are also used to hide the extraction and application of evaluation contexts, which are “meta-level” operations, carrying no computational meaning; in Section 9, equations are also used to decompose the evaluation tasks into their corresponding subtasks; finally, in Sections 7 and 10, equations of associativity and commutativity are used to achieve, respectively, modularity of language definitions, and true concurrency in chemical-soup-like computations. The point in all these cases is always the same: to achieve the right granularity of computations.

1.3. Outline of the paper

The remainder of this paper is organized as follows. Section 2 presents basic concepts of rewriting logic and recalls its deduction rules and its relationship with equational logic and term rewriting. Section 3 introduces a simple imperative operations, carrying no computational meaning; in Section 9, equations are also used to hide the extraction and application of evaluation contexts, which are “meta-level” operations, carrying no computational meaning; in Section 9, equations are also used to decompose the evaluation tasks into their corresponding subtasks; finally, in Sections 7 and 10, equations of associativity and commutativity are used to achieve, respectively, modularity of language definitions, and true concurrency in chemical-soup-like computations. The point in all these cases is always the same: to achieve the right granularity of computations.

2. Rewriting logic

Rewriting logic [50] 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 [48]. 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 [56,57]).

Two key points to explain are: (i) how rewriting 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 \( R = (\Sigma,E,R) \) 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

\[
(\forall X) \ t \rightarrow t' \text{ if } \left( \bigwedge_i u_i = u_i' \right) \land \left( \bigwedge_j w_j \rightarrow w_j' \right)
\]

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 \( (\Sigma,\mathcal{R}) \) can likewise be faithfully represented as the rewrite theory \( (\Sigma,\emptyset,\mathcal{R}) \).

Of course, if the equations of an equational theory \( (\Sigma,E) \) are confluent, there is another useful representation, namely, as the rewrite theory \( (\Sigma,\emptyset,\overline{E}) \), where \( \overline{E} \) 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 rewriting logic. In rewriting logic a rewrite theory \( \mathcal{R} = (\Sigma,E,\mathcal{R}) \) 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 \( \mathcal{R} \). The inference system of rewriting 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 rewriting logic deduction coincide.

Rewriting logic deduction

The inference rules below assume a typed setting, in which \( (\Sigma,E) \) is a membership equational theory [52] 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 $R = (\Sigma, E, \mathcal{R})$, the sentences that $R$ 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' 

- **Congruence.** For each $f : t_1, \ldots, t_n \rightarrow s$ in $\Sigma$, with $t_i \in T_\Sigma(X)_{s_i}, 1 \leq i \leq n$, and with $t'_j \in T_\Sigma(X)_{s'_j}, 1 \leq j \leq m$,

$$
(\forall X) t_1 \rightarrow t'_1 \quad \ldots \quad (\forall X) t_n \rightarrow t'_n

(\forall X) f(t_1, \ldots, t_j, \ldots, t_m, \ldots, t_n) \rightarrow f(t_1, \ldots, t'_j, \ldots, t'_m, \ldots, t_n)
$$

- **Replacement.** For each $\theta : X \rightarrow T_\Sigma(Y)$ and for each rule in $R$ of the form $(\forall X) t \rightarrow t'$ if $\left(\bigwedge_i u_i = u'_i\right) \land \left(\bigwedge_j w_j \rightarrow w'_j\right)$

$$
\left(\bigwedge_i (\forall Y) \theta(u_i) \rightarrow \theta'(u'_i)\right) \land \left(\bigwedge_j (\forall Y) \theta(w_j) = \theta(w'_j)\right)

\frac{(\forall Y) \theta(t) \rightarrow \theta'(t')}{(\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

\frac{(\forall X) t_1 \rightarrow t_3}{(\forall X) t_1 \rightarrow t_3}
$$

We can visualize the above inference rules as in Fig. 1.

The notation $R \vdash t \rightarrow t'$ states that the sequent $t \rightarrow t'$ is provable in the theory $R$ 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 rewriting logic [15] allows rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup A, \mathcal{L}, \phi)$, 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.\footnote{We will not consider this general version. The interested reader is referred to [15].}

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 $\rightarrow^1$ is defined by allowing either one or no applications of Replacement in the last condition. Similarly, one can define relations $\rightarrow^n$ (or $\rightarrow^{\leq n}$) by controlling the number of applications of the Transitivity rule. However, it should be noted that rewriting 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 rewriting 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 [92], 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: Asp+Sof [87], CafeOBJ [28], Elan [9], Maude [22], OBJ [38], and Stratego [93]. Some of these engines can achieve remarkable speeds on today’s machines, in the order of tens of millions of rewrite steps per second.

3. A simple imperative language

To illustrate the various operational semantics styles, we have chosen a small imperative language having arithmetic and boolean expressions with side effects (increment expression), short-circuited boolean operations, assignment, conditional, while loop, sequential composition, blocks, and halt. The syntax of the language is depicted in Table 1.

The semantics of $++x$ is that of incrementing the value of $x$ in the store and then returning the new value. The increment is done at the moment of evaluation, not after the end of the statement as in C/C++. Also, we assume short-circuit semantics for boolean operations.
This BNF syntax is entirely equivalent to an algebraic order-sorted signature having one (mixfix) operation definition per production, terminals giving the name of the operation and non-terminals the arity. For example, the production defining if-then-else can be seen as an algebraic operation

\[
\text{if\_then\_else~ : BExp} \times \text{Stmt} \times \text{Stmt} \rightarrow \text{Stmt}
\]

We will use the following conventions for variables throughout the remainder of the paper: \(X \in \text{Var}, A \in AExp, B \in BExp, St \in \text{Stmt}, P \in \text{Pgm}, i \in \text{Int}, T \in \text{Bool} = \{\text{true, false}\},\) any of them primed or indexed.

The base sections will use this simple language and will present definitions in various operational semantics styles (big-step, small-step SOS, MSOS, reduction using evaluation contexts, continuation-based, and Cham), as well the corresponding RLS representation of each definition. 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 [44,70,65,96,8] for further details on the described operational semantics styles.

We assume equational definitions for basic operations on booleans and integers, and assume that any other theory defined in the rest of this paper includes them. One of the reasons why we wrapped booleans and integers in the syntax is precisely to distinguish them from the corresponding values, and thus to prevent the "builtin" equations from reducing expressions like \(3 + 5\) directly in the syntax (we wish to have full control over the computational granularity of the language), since our RLS representations aim to have the same computational granularity of each of the different styles represented.

### 4. Store

Unlike in various operational semantics, which usually abstract stores as functions, in rewriting logic we explicitly define the store as an algebraic datatype: a store is a set of bindings from variables to values, together with two operations on them, one for retrieving a value, another for setting a value. We show that well-formed stores correspond to partially defined functions. Having this abstraction in place, we can regard them as functions for all practical purposes from now on.

To define the store, we assume a pairing "binding" constructor \(\_ \mapsto \_\), associating values to variables,\(^2\) and an associative and commutative union operation \(\_ \cup \_\) with \(\emptyset\) as its identity to put together such bindings. The equational definition \(\mathcal{E}_{\text{Store}}\) of operations \(\_ \mapsto \_\) to retrieve the value of a variable in the store and \(\_ \cup \_\) to update the value of a variable is given by the following equations, that operate modulo the associativity and commutativity of \(\_ \cup \_\):

\[
\begin{align*}
(S \mapsto I)(X) &= I \\
(S \mapsto I)(X') &= S(X') \quad \text{if} \ X \neq X' \\
(S \mapsto I)(X \leftarrow I') &= S \mapsto I' \\
(S \mapsto I)(X' \leftarrow I') &= S(X' \leftarrow I') \mapsto I \quad \text{if} \ X \neq X' \\
\emptyset(X \leftarrow I) &= X \mapsto I
\end{align*}
\]

Note the \(X \neq X\) 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 variables, for example. Since these definitions are equational, from a rewriting logic semantic point of view they are invisible: transitions are performed \emph{modulo} these equations. This way we can maintain a coarser computational granularity, while making use of auxiliary functions defined using equations. Although it might seem that, by using built-ins as integers and names, one cannot guarantee the existence of the initial model, notice that all the "built-ins" appearing in these definitions (names, booleans, integers) are definable as initial models of corresponding equational theories. And indeed, when performing formal proofs, one will make use of these equational definitions of the so-called built-ins. A store \(s\) is \emph{well formed} if \(\mathcal{E}_{\text{Store}} \vdash s = s_1 \mapsto i_1 \ldots s_n \mapsto i_n\) for some \(s_j \in \text{Var}\) and \(i_j \in \text{Int},\) for all \(1 \leq j \leq n,\) such that \(x_i \neq x_j\) for any \(i \neq j.\) We say that a store \(s\) is equivalent to a finite partial function \(\sigma : \text{Var} \rightarrow \text{Int},\) written \(s \preceq \sigma,\) if \(s\) is well formed and behaves as \(\sigma,\) that is, if for any \(x \in \text{Var}, i \in \text{Int}, \sigma(x) = i\) \iff \(\mathcal{E}_{\text{Store}} \vdash (s[X] = i).\) We recall that, given a store-function \(\sigma,\) \(\sigma[i/x]\) is defined as the function mapping \(x\) to \(i\) and other variables \(y\) to \(\sigma(y).\)

\(^2\) In general, one would have both an environment, and a store, with variables mapped to locations in the environment, and locations mapped to values in the store. However, for the sake of brevity, and given the simplicity of our example language, we do not use environments and map variables directly to values in the store.
Proposition 1. Let \( x, x' \in \text{Var}, i, i' \in \text{Int}, s, s' \in \text{Store} \) and finite partial functions \( \sigma, \sigma' : \text{Var} \to \text{Int} \).

1. \( \emptyset \bowtie \bot \) where \( \bowtie \) is the function undefined everywhere.
2. \( (s \ x \mapsto i) \bowtie \sigma \) implies that \( s \bowtie \sigma[\bot /x] \) where \( \sigma[\bot /x] \) is defined as \( \sigma \) restricted to \( \text{Dom}(\sigma) \setminus \{x\} \).
3. If \( s \bowtie \sigma \) then also \( s[i/x] \bowtie \sigma[i/x] \).

Proof. (1) Trivial, since \( \mathcal{E}_{\text{Store}} \nvdash \emptyset[x] = i \) for any \( x \in \text{Var} \in \text{Int} \).
(2) Let \( \sigma' \) be such that \( s \bowtie \sigma' \). We will prove that \( \text{Dom}(\sigma') = \text{Dom}(\sigma) \setminus \{x\} \) and for any \( x' \in \text{Dom}(\sigma') \), \( \sigma'(x') = \sigma(x) \). Consider an arbitrary \( x' \). If \( x' = x \), then \( \mathcal{E}_{\text{Store}} \nvdash s[x'] = i' \) for any \( i' \), since otherwise we would have \( \mathcal{E}_{\text{Store}} \vdash s = s' x \mapsto i' \) which contradicts the well formedness of \( s \mapsto i' \); therefore, \( \sigma' \) is not defined on \( x' \). If \( x' \neq x \), then \( \mathcal{E}_{\text{Store}} \vdash s[x'] = (s \ x \mapsto i)[x'] \), therefore \( \sigma' \) is defined on \( x' \) iff \( \sigma \) is defined on \( x' \), and if so \( \sigma'(x') = \sigma(x') \).
(3) Suppose \( s \bowtie \sigma \). We distinguish two cases—\( \sigma' \) is defined on \( x \) or if it is not. If it is, then let us say that \( \sigma(x) = i' \); in that case we must have that \( \mathcal{E}_{\text{Store}} \vdash s[x] = i' \) which can only happen if \( \mathcal{E}_{\text{Store}} \vdash s = s' x \mapsto i' \), whence \( \mathcal{E}_{\text{Store}} \vdash s[i/x] = s' x \mapsto i. \) Let \( x' \) be an arbitrary variable in \( \text{Var} \). If \( x' = x \) then

\[ \mathcal{E}_{\text{Store}} \vdash (s[x \mapsto i])[x'] = (s' x \mapsto i)[x'] = i \]

If \( x' \neq x \) then

\[ \mathcal{E}_{\text{Store}} \vdash (s[x \mapsto i])[x'] = (s' x \mapsto i)[x'] = s'[x'] = (s' x \mapsto i')[x'] = s'[x'] \]

If \( \sigma \) is not defined for \( x \), it means that \( \mathcal{E}_{\text{Store}} \nvdash s[x] = i \) for any \( i \), whence \( \mathcal{E}_{\text{Store}} \nvdash s = s' x \mapsto i \). If \( \mathcal{E}_{\text{Store}} \vdash s = \emptyset \) then we are done, since \( \mathcal{E}_{\text{Store}} \vdash (x \mapsto i)[x'] = i' \) iff \( x = x' \) and \( i = i' \). If \( \mathcal{E}_{\text{Store}} \nvdash s = \emptyset \), it must be that \( \mathcal{E}_{\text{Store}} \vdash s = x_1 \mapsto i_1 \ldots x_n \mapsto i_n \) with \( x_1 \neq x \). This leads to \( \mathcal{E}_{\text{Store}} \vdash s[x \mapsto i] = \cdots = (x_1 \mapsto i_1 \ldots x_n \mapsto i_n)(x \mapsto i)(x_{i+1} \mapsto i_{i+1} \ldots x_n \mapsto i_n) = \cdots = \emptyset(x \mapsto i)s = (x \mapsto i)s = s(x \mapsto i). \)

In the following, we will use symbols \( S, S', S_1, \ldots \), to denote variables of type \( \text{Store} \).

5. Big-step operational semantics

Introduced as natural semantics in [44], also named relational semantics in [60], 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 semantics can be easily represented within rewriting logic. For example, consider the big-step rule defining integer division

\[
\frac{\langle A_1, \sigma \rangle \Downarrow (l_1, \sigma_1), \langle A_2, \sigma_1 \rangle \Downarrow (l_2, \sigma_2)}{\langle A_1 / A_2, \sigma \rangle \Downarrow (l_1, \text{Int}l_2, \sigma_2)}, \quad \text{if } l_2 \neq 0
\]

This rule can be automatically translated into the rewrite rule

\[ \langle A_1 / A_2, S \rangle \to \langle l_1, \text{Int}l_2, S_2 \rangle \text{ if } \langle A_1, S \rangle \to \langle l_1, S_1 \rangle \land \langle A_2, S_1 \rangle \to \langle l_2, S_2 \rangle \land l_2 \neq 0 \]

The complete big-step operational semantics definition for our simple language, except its \texttt{halt} statement (which is discussed at the end of this section), which we call \texttt{BigStep}, is presented in Table 2. We choose to exclude from the presentation the semantics for constructs entirely similar to the ones presented, such as “\texttt{-},” “\texttt{+},” “\texttt{/},” and “\texttt{\%}.” To give a rewriting logic theory for the big-step semantics, one needs to first define the various configuration constructs, which are assumed by default in \texttt{BigStep}, as corresponding operations extending the signature. Then one can define the rewrite theory \( R_{\text{BigStep}} \) corresponding to the big-step operational semantics \texttt{BigStep} entirely automatically as shown by Table 3. Note that, because the rewriting relation is reflexive, we did not need to add the reflexivity rules for boolean and integer values.

Due to the one-to-one correspondence between big-step rules in \texttt{BigStep} and rewrite rules in \( R_{\text{BigStep}} \), it is easy to prove by induction on the length of derivations the following result:

Proposition 2. For any \( p \in \text{Pgm} \) and \( i \in \text{Int} \), the following are equivalent:

1. \( \texttt{BigStep} \vdash \langle p \rangle \Downarrow (i) \)
2. \( R_{\text{BigStep}} \vdash \langle p \rangle \to^1 (i) \)

Proof. A first thing to notice is that, since all rules involve configurations, rewriting can only occur at the top, thus the general application of term rewriting under contexts is disabled by the definitional style. Another thing to notice here is
that all configurations in the right hand sides are normal forms, thus the transitivity rule for rewriting logic also becomes inapplicable. Suppose \( s \in \text{Store} \) and \( \sigma : \text{Var} \rightarrow \text{Int} \) such that \( s \simeq \sigma \). We prove the following statements:

(1) \( \text{BigStep} \vdash (a, \sigma) \downarrow (i, \sigma') \) iff \( R_{\text{BigStep}} \vdash (a, s) \rightarrow (i, s') \) and \( s' \simeq \sigma' \),
    for any \( a \in \text{AExp}, i \in \text{Int}, \alpha : \text{Var} \rightarrow \text{Int} \) and \( \sigma' \in \text{Store} \).

(2) \( \text{BigStep} \vdash (b, \sigma) \downarrow (t, \sigma') \) iff \( R_{\text{BigStep}} \vdash (b, s) \rightarrow (t, s') \) and \( s' \simeq \sigma' \),
    for any \( b \in \text{AExp}, t \in \text{Bool}, \alpha : \text{Var} \rightarrow \text{Int} \) and \( \sigma' \in \text{Store} \).

(3) \( \text{BigStep} \vdash (\text{st}, \sigma) \downarrow (\sigma') \) iff \( R_{\text{BigStep}} \vdash (\text{st}, s) \rightarrow (\sigma') \) and \( s' \simeq \sigma' \),
    for any \( s \in \text{Stmt}, \alpha : \text{Var} \rightarrow \text{Int} \) and \( \sigma' \in \text{Store} \).

(4) \( \text{BigStep} \vdash (p) \downarrow (i) \) iff \( R_{\text{BigStep}} \vdash (p) \rightarrow (i, \alpha) \),
    for any \( p \in \text{Pgm} \) and \( i \in \text{Int} \).

Each can be proved by induction on the size of the derivation tree. To avoid lengthy and repetitive details, we discuss the corresponding proof of only one language construct in each category:

(1) \( \text{BigStep} \vdash (x\text{\texttt{++}}, \sigma) \downarrow (i, \sigma'[i/x]) \) iff

\[
i = \sigma(X) + 1 \text{ iff } \epsilon_{\text{Store}} \subseteq R_{\text{BigStep}} \vdash i = s[X] + 1 \text{ iff } \epsilon_{\text{Store}} \subseteq R_{\text{BigStep}} \vdash (x\text{\texttt{++}}, s) \rightarrow (i, s[X \leftarrow i]).
\]

This completes the proof, since \( s[X \leftarrow i] \simeq \sigma[i/x] \), by 3 in Proposition 1.
The only apparent difference between BigStep and $\mathcal{R}_{\text{BigStep}}$ is the different notational conventions they use. However, as the above theorem shows, there is a one-to-one correspondence also between their corresponding “computations” (or executions, or derivations). Therefore, $\mathcal{R}_{\text{BigStep}}$ actually is the big-step operational semantics BigStep, not an “encoding” of it. Note that, in order to be faithfully equivalent to BigStep computationally, $\mathcal{R}_{\text{BigStep}}$ lacks the main strength of rewriting logic theory that makes it an appropriate formalism for concurrency, namely, that rewrite rules can apply under any context and in parallel (here all rules are syntactically constrained so that they can only apply at the top, sequentially).

**Strengths.** Big-step operational semantics allows straightforward recursive definition. It can be easily and efficiently interpreted in any recursive, functional, or logical framework. It is particularly useful for defining type systems.

**Weaknesses.** 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 specified...

---

**Table 3**

$\mathcal{R}_{\text{BigStep}}$ rewriting logic theory.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle X,S \rangle \rightarrow \langle S[X],S \rangle$</td>
<td>$\langle X,S \rangle \rightarrow \langle l,S[l] \rangle$ if $l = S[X]+1$</td>
</tr>
<tr>
<td>$\langle A_1 \triangleleft A_2,S \rangle \rightarrow \langle l_1 + \text{Int} l_2,S \rangle$</td>
<td>$\langle A_1,S \rangle \rightarrow \langle l_1,S_1 \rangle \land \langle A_2,S_1 \rangle \rightarrow \langle l_2,S_2 \rangle$</td>
</tr>
<tr>
<td>$\langle B_1 \triangleleft B_2,S \rangle \rightarrow \langle T,S \rangle$</td>
<td>$\langle B_1,S \rangle \rightarrow \langle T,S_1 \rangle \land \langle B_2,S_1 \rangle \rightarrow \langle T,S_2 \rangle$</td>
</tr>
<tr>
<td>$\langle \text{not} R,S \rangle \rightarrow \langle \text{not}(T,S) \rangle$</td>
<td>$\langle B,S \rangle \rightarrow \langle T,S' \rangle$</td>
</tr>
</tbody>
</table>

$(2)$ BigStep $\vdash \langle b_1 \text{ and } b_2,\sigma \rangle \downarrow (t,\sigma')$ iff

- $\text{BigStep} \vdash \langle b_1,\sigma \rangle \downarrow (\text{false},\sigma')$ and $t = \text{false}$
- or $\text{BigStep} \vdash \langle b_1,\sigma \rangle \downarrow (\text{true},\sigma'')$ and $\text{BigStep} \vdash \langle b_2,\sigma'' \rangle \downarrow (t,\sigma')$

$(3)$ BigStep $\vdash \langle \text{while } b \text{ st},\sigma \rangle \downarrow (\sigma')$ iff

- $\text{BigStep} \vdash \langle b,\sigma \rangle \downarrow (\text{false},\sigma')$
- or $\text{BigStep} \vdash \langle b,\sigma \rangle \downarrow (\text{true},\sigma_1)$ and $\text{BigStep} \vdash \langle \text{st},\sigma_1 \rangle \downarrow (\sigma_2)$
- and $\text{BigStep} \vdash \langle \text{while } b \text{ st},\sigma_2 \rangle \downarrow (\sigma')$ iff

$(4)$ BigStep $\vdash \langle \text{st},\sigma \rangle \downarrow (i,\sigma)$ and $\text{BigStep} \vdash \langle a,\sigma \rangle \downarrow (i,\sigma')$ iff

This completes the proof. \qed
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; consider, for example, adding halt to the above definition—one needs to add a special configuration \(halt(l)\), and the following rules:

\[
\begin{align*}
  &\langle\text{halt } A, S\rangle \rightarrow \langle\text{halt}(l)\rangle \\
  &\langle St_1, St_2, S\rangle \rightarrow \langle\text{halt}(l)\rangle \\
  &\langle\text{while } B, St, S\rangle \rightarrow \langle\text{halt}(l)\rangle \\
  &\langle St, A, S\rangle \rightarrow \langle I\rangle \\
  &\langle St, \emptyset\rangle \rightarrow \langle\text{halt}(l)\rangle
\end{align*}
\]

6. Small-step operational semantics

Introduced by Plotkin in [70], also called transition semantics or reduction semantics, small-step semantics captures the notion of one computational step.

One inherent technicality involved in capturing small-step operational semantics as a rewrite theory in a one-to-one notational and computational correspondence is that the rewriting relation is by definition transitive, while the small-step

Table 4

The SmallStep language definition.

<table>
<thead>
<tr>
<th>Types of configurations: ((\text{AExp,Store}), (\text{BExp,Store}), (\text{Stmt,Store}), (\text{Pgm,Store}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle X, \sigma \rangle \rightarrow \langle \sigma(X), \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle ++X, \sigma \rangle \rightarrow \langle I, \sigma[I/X]\rangle) if (I = \sigma(X) + 1)</td>
</tr>
<tr>
<td>(\langle A_1, \sigma \rangle \rightarrow \langle A'_1, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle A_2, \sigma \rangle \rightarrow \langle A'_2, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle A_1 + A_2, \sigma \rangle \rightarrow \langle A'_1 + A'_2, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle I_1 + I_2, \sigma \rangle \rightarrow \langle I_1 + \text{Int} I_2, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle B_1, \sigma \rangle \rightarrow \langle B'_1, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle B_2, \sigma \rangle \rightarrow \langle B'_2, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle B_1 \leq \text{Int} I_2, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle B_1 \text{ and } B_2, \sigma \rangle \rightarrow \langle B'_1 \text{ and } B'_2, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle B, \sigma \rangle \rightarrow \langle B', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{not } B, \sigma \rangle \rightarrow \langle \text{not } B', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{true and } B_2, \sigma \rangle \rightarrow \langle B_2, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{false and } B_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{false}, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle A, \sigma \rangle \rightarrow \langle A', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle X := A, \sigma \rangle \rightarrow \langle X := A', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle X := I, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[I/X] \rangle)</td>
</tr>
<tr>
<td>(\langle \text{skip}, \sigma \rangle \rightarrow \langle \text{skip}, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle St_1, \sigma \rangle \rightarrow \langle St_1', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle St_2, \sigma \rangle \rightarrow \langle St_2', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle St, A, S \rangle \rightarrow \langle St', A', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle A, \sigma \rangle \rightarrow \langle A', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{not } A, \sigma \rangle \rightarrow \langle \text{not } A', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{true and } A_2, \sigma \rangle \rightarrow \langle A_2, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{false and } A_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{false}, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle B, \sigma \rangle \rightarrow \langle B', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{if } B \text{ then } St_1 \text{ else } St_2, \sigma \rangle \rightarrow \langle \text{if } B' \text{ then } St_1 \text{ else } St_2, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{if } \text{true} \text{ then } St_1 \text{ else } St_2, \sigma \rangle \rightarrow \langle St_1, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{if } \text{false} \text{ then } St_1 \text{ else } St_2, \sigma \rangle \rightarrow \langle St_2, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{while } B, St, \sigma \rangle \rightarrow \langle \text{if } B \text{ then } St, \text{while } B, St \text{ else skip}, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle St, \sigma \rangle \rightarrow \langle St', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle A, \sigma \rangle \rightarrow \langle A', \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{skip}, \sigma \rangle \rightarrow \langle \text{skip}, \sigma' \rangle)</td>
</tr>
<tr>
<td>(\langle \text{eval}(P), \sigma \rangle \rightarrow \langle I\rangle)</td>
</tr>
</tbody>
</table>
relation is not transitive (its transitive closure can be defined a posteriori). Therefore, we need to devise a mechanism to "inhibit" rewriting logic's transitive and uncontrolled application of rules. An elegant way to achieve this is to view a small-step as a modifier of the current configuration. Specifically, we consider "·" to be a modifier on the configuration which gives more flexibility to the specification—for example, one can specify that one wants two steps simply by putting two dots in front of the other "tagged" with the modifier. However, allowing "·" to be a modifier rather than a part of a configuration gives more transitivity to the specification—e.g., one can specify that one wants two steps simply by putting two dots in front of the configuration.

A similar technique is proposed in [55], but there two different types of configurations are employed, one standard and the other "tagged" with the modifier. However, allowing "·" to be a modifier rather than a part of a configuration gives more flexibility to the specification—for example, one can specify that one wants two steps simply by putting two dots in front of the configuration.

The complete small-step operational semantics definition for our simple language except its halt statement (which is discussed at the end of this section), which we call SmallStep, is presented in Table 4. The corresponding small-step rewriting logic theory \( R_{\text{SmallStep}} \) is given in Table 5. The language described here does not involve labels on rules like in the SOS of concurrent systems. For that, one would take an approach similar to that presented in Section 7, that is, pushing the labels back into the configurations.

\[ (X, S) \rightarrow (S[X], S) \]
\[ (X := A, S) \rightarrow (X := A', S) \]
\[ (S[1], S[2]) \rightarrow (S[1'], S[2']) \]

Table 5

\[ R_{\text{SmallStep}} \] rewriting logic theory.

\[ \text{eval}(P) = \text{smallstep}(P, \bot) \]
\[ \text{smallstep}(P, S) = \text{smallstep}(P, S) \]
\[ \text{smallstep}(\text{skip}, S) = I \]

\[ \langle A_1, S \rangle \rightarrow \langle A_1', S' \rangle \]
\[ \langle A_1 + A_2, S \rangle \rightarrow \langle A_1' + A_2, S' \rangle \] is translated, again automatically, into a rewriting logic rule, e.g.,

\[ \langle A_1 + A_2, S \rangle \rightarrow \langle A_1' + A_2, S' \rangle \] if \( \langle A_1, S \rangle \rightarrow \langle A_1', S' \rangle \)
As for big-step semantics, the rewriting under context deduction rule for rewriting logic is again inapplicable, since all rules act at the top, on configurations. However, in SmallStep it is not the case that all right hand sides are normal forms (this actually is a key feature of small-step semantics). The "→" operator introduced in \( R_{\text{SmallStep}} \) prevents the unrestricted application of transitivity, and can be regarded as a token given to a configuration to allow it to change to the next step. We use transitivity at the end (rules for smallstep) to obtain the transitive closure of the small-step relation by specifically giving tokens to the configuration until it reaches a normal form.

Again, there is a direct correspondence between SOS-style rules and rewriting rules, leading to the following result, which can also be proved by induction on the length of derivations:

**Proposition 3.** For any \( p \in \text{Pgm}, \sigma, \sigma' : \text{Var} \xrightarrow{\sigma} \text{Int} \) and \( s \in \text{Store} \) such that \( s \simeq \sigma \), the following are equivalent:

1. \( \text{SmallStep} \vdash (p, \sigma) \rightarrow (p', \sigma') \), and
2. \( R_{\text{SmallStep}} \vdash (p, s) \rightarrow^1 (p', s') \) and \( s' \simeq \sigma' \).

Moreover, the following are equivalent for any \( p \in \text{Pgm} \) and \( i \in \text{Int} \):

1. \( \text{SmallStep} \vdash (p, \bot) \rightarrow^* (\text{skip}, i, \sigma) \) for some \( \sigma : \text{Var} \xrightarrow{\sigma} \text{Int} \), and
2. \( R_{\text{SmallStep}} \vdash \text{eval}(p) \rightarrow i. \)

**Proof.** As for big-step, we split the proof into four cases, by proving for each syntactical category the following facts (suppose \( s \in \text{Store}, \sigma : \text{Var} \xrightarrow{\sigma} \text{Int}, s \simeq \sigma \)):

1. \( \text{SmallStep} \vdash (a \sigma) \rightarrow (a' \sigma') \) iff \( R_{\text{SmallStep}} \vdash (a, s) \rightarrow^1 (a' s') \) and \( s' \simeq \sigma' \), for any \( a, a' \in \text{AExp}, \sigma' : \text{Var} \xrightarrow{\sigma} \text{Int} \) and \( s' \in \text{Store} \).
2. \( \text{SmallStep} \vdash (b \sigma) \rightarrow (b' \sigma') \) iff \( R_{\text{SmallStep}} \vdash (b, s) \rightarrow^1 (b' s') \) and \( s' \simeq \sigma' \), for any \( b, b' \in \text{BExp}, \sigma' : \text{Var} \xrightarrow{\sigma} \text{Int} \) and \( s' \in \text{Store} \).
3. \( \text{SmallStep} \vdash (s \sigma) \rightarrow (s' \sigma') \) iff \( R_{\text{SmallStep}} \vdash (s, s') \rightarrow^1 (s' s') \) and \( s' \simeq \sigma' \), for any \( s, s' \in \text{Stmt}, \sigma' : \text{Var} \xrightarrow{\sigma} \text{Int} \) and \( s' \in \text{Store} \).
4. \( \text{SmallStep} \vdash (p \sigma) \rightarrow (p' \sigma') \) iff \( R_{\text{SmallStep}} \vdash (p, s) \rightarrow^1 (p' s') \) and \( s' \simeq \sigma' \), for any \( p, p' \in \text{Pgm}, \sigma' : \text{Var} \xrightarrow{\sigma} \text{Int} \) and \( s' \in \text{Store} \).

These equivalences can be shown by induction on the size of the derivation tree. Again, we only show one example per category:

1. \( \text{SmallStep} \vdash (a_1 + a_2 \sigma) \rightarrow (a_1 + a_2' \sigma') \) iff \( a_1 = i \) and \( \text{SmallStep} \vdash (a_2 \sigma) \rightarrow (a_2' \sigma') \) iff \( \text{SmallStep} \vdash (a_1 + a_2, s) \rightarrow^1 (a_1' s') \) and \( s' \simeq \sigma' \).
2. \( \text{SmallStep} \vdash (\text{not} \ true, \sigma) \rightarrow (\text{false}, \sigma') \) iff \( \text{SmallStep} \vdash (\text{not} \ true, s) \rightarrow^1 (\text{false}, s') \).
3. \( \text{SmallStep} \vdash (s_1; s_2 \sigma) \rightarrow (s_1', s_2', s' \sigma') \) iff \( \text{SmallStep} \vdash (s_1, s_2 \sigma) \rightarrow (s_1', s_2', s' \sigma') \). Let us now move to the second equivalence. For this proof let \( \rightarrow^1 \) be the restriction of \( R_{\text{SmallStep}} \) relation \( \rightarrow \) to those pairs which can be provable by exactly applying \( n-1 \) times the Transitivity rule if \( n > 0 \), or Reflexivity for \( n = 0 \). We first prove the following more general result (suppose \( p \in \text{Pgm}, \sigma : \text{Var} \xrightarrow{\sigma} \text{Int} \) and \( s \in \text{Store} \) such that \( s \simeq \sigma \)):

\[
\text{SmallStep} \vdash (p, \sigma) \rightarrow^1 (p', \sigma') \quad \text{iff} \quad R_{\text{SmallStep}} \vdash \text{smallstep}(p, s) \rightarrow^0 \text{smallstep}(p, s') \quad \text{and} \quad s' \simeq \sigma',
\]

by induction on \( n \). If \( n = 0 \) then \( (p, \sigma) \rightarrow (p', \sigma') \) and since \( R_{\text{SmallStep}} \vdash \text{smallstep}(p, s) = \text{smallstep}(p, s) \) we are done. If \( n > 0 \), we have that

\[
\text{SmallStep} \vdash (p, \sigma) \rightarrow^0 (p', \sigma') \quad \text{iff} \quad \text{SmallStep} \vdash (p, \sigma) \rightarrow^0 (p_1, \sigma_1) \quad \text{and} \quad \text{SmallStep} \vdash (p_1, \sigma_1) \rightarrow^0 \cdots \rightarrow^{n-1} (p', \sigma') \quad \text{iff} \quad R_{\text{SmallStep}} \vdash (p, s) \rightarrow (p_1, s_1) \quad \text{and} \quad s_1 \simeq \sigma_1 \quad (\text{by } 1)
\]
and \( R_{\text{SmallStep}} \vdash \text{smallstep}(p_1, s_1) \rightarrow^{n-1} \text{smallstep}((p', s')) \) and \( s' \simeq \sigma' \) iff

(by the induction hypothesis)

\[ R_{\text{SmallStep}} \vdash \text{smallstep}((p, s)) \rightarrow^1 \text{smallstep}(p_1, s_1) \] and \( s_1 \simeq \sigma_1 \)

\[ R_{\text{SmallStep}} \vdash \text{smallstep}(p, s) \rightarrow^n \text{smallstep}((p', s')) \) and \( s' \simeq \sigma' \) iff

\[ R_{\text{SmallStep}} \vdash \text{smallstep}(p, s) \rightarrow \text{smallstep}((p, s)). \]

We are done, since \( R_{\text{SmallStep}} \vdash \text{smallstep}(p, s) = \text{smallstep}((p, s)). \)

Finally, \( \text{SmallStep} \vdash (p, \bot) \rightarrow^* (\text{skip}, \sigma) \) iff \( R_{\text{SmallStep}} \vdash \text{smallstep}(p, \bot) \rightarrow \text{smallstep}((\text{skip}, i, \sigma)), s \simeq \sigma; \) the rest follows from \( R_{\text{SmallStep}} \vdash \text{eval}(p) = \text{smallstep}(p, \bot) \) and \( R_{\text{SmallStep}} \vdash \text{smallstep}((\text{skip}, i, \sigma)) = i. \)

**Strengths.** Small-step operational semantics precisely defines the notion of one computational step. It stops at errors, pointing them out. It is easy to trace and debug. It gives interleaving semantics for concurrency.

**Weaknesses.** Each small step does the same amount of computation as a big-step in finding the next redex. 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. One of the reasons for introducing SOS was 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. One of the reasons for introducing SOS was that abstract machines need to introduce new syntactic constructs to decompose the abstract syntax tree, while SOS would and should only work by modifying the structure of the program. We argue that this is not entirely accurate: for example, one needs to have the syntax of boolean values if one wants to have boolean expressions, and needs an \( \forall \) mechanism in the above definition to evaluate \( \forall \). The fact that these features are common in programming languages does not mean that the languages which don’t want to allow them should be despised. It is still hard to deal with control—for example, consider adding halt to this language. One cannot simply do it as for other ordinary statements: instead, one has to add a corner case (additional rule) to each statement, as shown below:

\[
\begin{align*}
&\text{Unobs}(\sigma = \sigma_0, \sigma' = \sigma_0) \\
&X : = l (\sigma = \sigma_0) \rightarrow^{0} \text{skip}
\end{align*}
\]

The above rule says that, if before the transition the store was \( \sigma_0 \), after the transition it will become \( \sigma_0[l/X] \), updating \( X \) by \( l \). The unobs predicate is used to express that the rest of the state does not change.

**Write-only fields** are used to record things whose values cannot be inspected before a transition such as emission of actions to the outside world (e.g., output, recording of the trace). Their names are always primed and they have a free monoid semantics—everything written on them is actually added at the end. A good example of the usage of write-only fields would

\[ \text{smallstep}(p, s) \rightarrow \text{smallstep}((p', s')). \]
Table 6
The MSOS language definition.

\[
\text{unobs}(\sigma, \ldots), \sigma(X) = I
\]
\[
\begin{array}{ll}
X & \xrightarrow{[\sigma = \sigma_0 \sigma'] = \sigma_0 \ldots, I = \sigma_0(X) + 1} I \\
A_1 & \xrightarrow{\cdot \sigma = \cdot \sigma'} A_1' \\
A_1 + A_2 & \xrightarrow{\cdot \sigma = \cdot \sigma'} A_1' + A_2' \\
A_1 + A_2 & \xrightarrow{\cdot \sigma = \cdot \sigma'} I_1 + A_2' \\
I_1 + I_2 & \xrightarrow{\cdot \sigma = \cdot \sigma'} I_1 + I_2'
\end{array}
\]

\[
\begin{array}{ll}
B_1 & \xrightarrow{\cdot \sigma = \cdot \sigma'} B_1' \\
B_1 & \xrightarrow{\cdot \sigma = \cdot \sigma'} B_1' \\
B_1 & \xrightarrow{\cdot \sigma = \cdot \sigma'} B_1' \\
B_1 & \xrightarrow{\cdot \sigma = \cdot \sigma'} B_1'
\end{array}
\]

\[
\begin{array}{ll}
A & \xrightarrow{\cdot \sigma = \cdot \sigma'} A' \\
X::A & \xrightarrow{\cdot \sigma = \cdot \sigma'} X::A' \\
\text{unobs}(\sigma = \sigma_0 \sigma' = \sigma_0 \ldots) \\
X::[\sigma = \sigma_0 \sigma'] & \xrightarrow{\text{skip}} \\
X::[] & \xrightarrow{\text{skip}} \\
\text{unobs}(\sigma = \sigma_0 \sigma' = \sigma_0 \ldots) \\
X::[\sigma = \sigma_0 \sigma'] & \xrightarrow{\text{skip}} \\
X::[] & \xrightarrow{\text{skip}}
\end{array}
\]

\[
\begin{array}{ll}
B & \xrightarrow{\cdot \sigma = \cdot \sigma'} B' \\
\text{if } B \text{ then } St_1 \text{ else } St_2 & \xrightarrow{\cdot \sigma'} \text{if } B' \text{ then } St_1 \text{ else } St_2 \\
\text{if true then } St_1 \text{ else } St_2 & \xrightarrow{\cdot \sigma'} \text{if true then } St_1 \text{ else } St_2 \\
\text{if false then } St_1 \text{ else } St_2 & \xrightarrow{\cdot \sigma'} \text{if false then } St_1 \text{ else } St_2 \\
\text{while } B \text{ St } \rightarrow \text{ if } B \text{ then } (\text{St}; \text{while } B \text{ St}) \text{ else skip}
\end{array}
\]

\[
\begin{array}{ll}
St & \xrightarrow{\cdot \sigma = \cdot \sigma'} St' \\
\text{Skit } A & \xrightarrow{\cdot \sigma = \cdot \sigma'} \text{Skit } A' \\
\text{Skit } A & \xrightarrow{\cdot \sigma = \cdot \sigma'} \text{Skit } A'
\end{array}
\]

be a rule for defining a print language construct

\[
\begin{array}{ll}
\text{unobs}(\text{out}' = (), \ldots) \\
\text{print}(I) & \xrightarrow{\text{out}' = (\ldots)} \text{skip}
\end{array}
\]

where "()" stand for monoid unit.

The state after this rule is applied will have the out field containing "LI", where the juxtaposition LI denotes the free monoid multiplication of L and I.

The MSOS description of the small-step SOS definition in Table 4 is given in Table 6 (we let A range over labels on transitions).

Because the part of the state not involved in a certain rule is hidden through the "..." notation, language extensions can be made modularly. Consider, for example, adding halt to the definition in Table 6. One possible way to do it is to follow the technique proposed in [65] for adding non-parametric abrupt termination, with some modifications to suit our needs to abruptly terminate the program with a value. For this, we add a write-only field in the record, say halt? having as arrows the monoid freely generated by integer numbers, along with a language construct stick to block the execution of the program. To "catch the halt signal" we extend the abstract syntax with a new construct, say program, applied to a top-level program. The first set of MSOS rules for halt are then
record notation in fields to be added.

using sets of fields and matching modulo associativity and commutativity. That is, the extensibility provided by the "modular is the record comprehension mechanism. A similar comprehension mechanism is achieved in rewriting logic by the program:

\[
\begin{align*}
A & \xrightarrow{\chi} A' \\
\text{halt } A & \xrightarrow{\chi} \text{ halt } A' \\
\text{unobs}(\text{halt}? = (0, \ldots)) & \\
\text{halt } I & \xrightarrow{\text{halt}? = (0, \ldots)} \text{ stuck} \\
I & \xrightarrow{\text{halt}? = (0, \ldots)} \text{ stuck} \\
\text{P} & \xrightarrow{\text{halt}? = (0, \ldots)} \text{ P'} \\
\text{program } P & \xrightarrow{\text{halt}? = (0, \ldots)} \text{ program skip.I} \\
\text{P} & \xrightarrow{\text{halt}? = (0, \ldots)} \text{ P'} \\
\text{program } P & \xrightarrow{\text{halt}? = (0, \ldots)} \text{ program } P'
\end{align*}
\]

An alternative to the above definition, which would not require the introduction of new syntax, is to make "halt?" a read-write field with possible values integers along with a default value nil and use an unobservable transition at top to terminate the program:

\[
\begin{align*}
A & \xrightarrow{\chi} A' \\
\text{halt } A & \xrightarrow{\chi} \text{ halt } A' \\
\text{unobs}(\text{halt}? = \text{nil}, \text{halt}? = \text{nil}, \ldots) & \\
\text{halt } I & \xrightarrow{\text{halt}? = \text{nil}, \text{halt}? = \text{nil}, \ldots} \text{ stuck} \\
\text{unobs}(\text{halt}? = \text{nil}, \text{halt}? = \text{nil}, \ldots) & \\
\text{I} & \xrightarrow{\text{halt}? = \text{nil}, \text{halt}? = \text{nil}, \ldots} \text{ stuck} \\
\text{P} & \xrightarrow{\text{halt}? = \text{nil}, \text{halt}? = \text{nil}, \ldots} \text{ skip.I} \\
\text{P} & \xrightarrow{\text{halt}? = \text{nil}, \text{halt}? = \text{nil}, \ldots} \text{ P'} \\
\text{program } P & \xrightarrow{\text{halt}? = \text{nil}, \text{halt}? = \text{nil}, \ldots} \text{ program } P'
\end{align*}
\]

However, since the last rule is based on observation of the state, the program is not forced to terminate as soon as halt is consumed (as was the case in the first definition), since in the case of non-determinism, for example, there might be other things which are still computable.

To give a faithful representation of MSOS definitions in rewriting logic, we here follow the methodology in [55]. Using the fact that labels describe changes from their source state to their destination state, one can move the labels back into the configurations. That is, a transition step \( P \xrightarrow{u} P' \) is modeled as a rewrite step \( \langle P, u^{\text{pre}} \rangle \rightarrow \langle P', u^{\text{post}} \rangle \), where \( u^{\text{pre}} \) and \( u^{\text{post}} \) are records describing the state before and after the transition. Notice again the use of the "\( \_ \)" operator to emulate small-steps by restricting transitivity. State records can be specified equationally as wrapping (using a constructor "[...]") a set of fields built from fields as constructors, using an associative and commutative concatenation operation "\( \_ \_ \)". Fields are constructed from state attributes; for example, the store can be embedded into a field by a constructor "\( \sigma : \_ \)".

Records \( u^{\text{pre}} \) and \( u^{\text{post}} \) are computed from \( u \) and \( v \) in the following way:

- For unobservable transitions, \( u^{\text{pre}} = u^{\text{post}} \); same applies for unobservable attributes in premises;
- Read-only fields of \( u \) are added to both \( u^{\text{pre}} \) and \( u^{\text{post}} \).
- Read-write fields of \( u \) are translated by putting the read part in \( u^{\text{pre}} \) and the (now unprimed) write part in \( u^{\text{post}} \). The assignment rule, for example, becomes
  \[
  \langle X := I, \sigma : S_0, W \rangle \rightarrow (\text{skip}, \sigma : S_0, X \equiv I, W)
  \]
  Notice that the "\( \_ \)" notation gets replaced by a generic field-set variable \( W \).
- Write-only fields \( i = v \) of \( u \) are translated as follows: \( i : L \), with \( L \) a fresh new variable, is added to \( u^{\text{pre}} \), and \( i : L \) is added to \( u^{\text{post}} \). For example, the print rule above becomes
  \[
  \langle \text{print}(I), \{\text{out} : L, W\} \rangle \rightarrow (\text{skip}, \{\text{out} : L, W\})
  \]
- When dealing with observable transitions, both state records meta-variables and \( \ldots \) operations are represented in \( u^{\text{pre}} \) by some variables, while in \( u^{\text{post}} \) by others. For example, the first rule defining addition in Table 6 is translated into
  \[
  \langle A_1 + A_2, R \rangle \rightarrow \langle A'_1 + A_2, R' \rangle \quad \text{if} \quad \langle A_1, R \rangle \rightarrow \langle A'_1, R' \rangle
  \]

The key thing to notice here is that modularity is preserved by this translation. What indeed makes MSOS definitions modular is the record comprehension mechanism. A similar comprehension mechanism is achieved in rewriting logic by using sets of fields and matching modulo associativity and commutativity. That is, the extensibility provided by the "\( \ldots \)" record notation in MSOS is here captured by associative and commutative matching on the \( W \) variable, which allows new fields to be added.
The relation between MSOS and RMSOS definitions assumes that MSOS definitions are in a certain normal form [55] and is made precise by the following theorem, strongly relating MSOS and modular rewriting semantics.

**Theorem 1** ([55]). For each normalized MSOS definition, there is a strong bisimulation between its transition system and the transition system associated to its translation in rewriting logic.

The above presented translation is the basis for the Maude–MSOS tool [19], which has been used to define and analyze complex language definitions, such as Concurrent ML [18].

Table 7 presents the rewrite theory corresponding to the MSOS definition in Table 6. The only new variable symbols introduced are \( R, R' \), standing for records, and \( W \) standing for the remainder of a record.

**Strengths.** As it is a framework on top of any operational semantics, it inherits the strengths of the semantics for which it is used; moreover, it adds to those strengths the important new feature of modularity. It is well known that SOS definitions are typically highly unmodular, so that adding a new feature to the language often requires the entire redefinition of the SOS rules.

**Weaknesses.** Control is still not explicit in MSOS, making combinations of control-dependent features (e.g., call/cc) impossible to specify [65, p. 223]. Also, MSOS still does not allow to capture the intended computational granularity of some defined language statements. For example, the desired semantics of “halt i” is “stop the execution with the result i”; unfortunately, MSOS, like its SOS ancestors, still needs to “propagate” the halting signal along the syntax all the way to the top.

### 8. Reduction semantics with evaluation contexts

Introduced in [96], also called context reduction, the evaluation contexts style improves over small-step definitional style 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

<table>
<thead>
<tr>
<th>Table 7</th>
<th>( R_{\text{MSOS}} ) rewriting logic theory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X : \sigma : S, W )</td>
<td>( I : \sigma : S, W ) ( \iff ) ( I = S[X] )</td>
</tr>
<tr>
<td>( \langle X, \sigma : S, W \rangle \rightarrow \langle I, \sigma : S, W \rangle ) ( \iff ) ( I = S[X] + 1 )</td>
<td></td>
</tr>
<tr>
<td>( A_1 \leftarrow A_2.R \rightarrow \langle A'_1 \leftarrow A'_2.R' \rangle ) ( \iff ) ( \langle A_2.R \rightarrow \langle A'_2.R' \rangle )</td>
<td></td>
</tr>
<tr>
<td>( I_1 \leftarrow I_2.R \rightarrow \langle I'_1 \leftarrow I'_2.R' \rangle ) ( \iff ) ( \langle I_2.R \rightarrow \langle I'_2.R' \rangle )</td>
<td></td>
</tr>
<tr>
<td>( B_1 \text{ and } B_2.R \rightarrow \langle B'_1 \text{ and } B'_2.R' \rangle ) ( \iff ) ( \langle B_2.R \rightarrow \langle B'_2.R' \rangle )</td>
<td></td>
</tr>
<tr>
<td>( \text{false and } B_2.R \rightarrow \text{false,R} ) ( \iff ) ( \langle B_2.R \rightarrow \langle B'_2.R' \rangle )</td>
<td></td>
</tr>
<tr>
<td>( \text{true and } B_2.R \rightarrow \text{true,R} ) ( \iff ) ( \langle B_2.R \rightarrow \langle B'_2.R' \rangle )</td>
<td></td>
</tr>
<tr>
<td>( X := A.R \rightarrow \langle X := A'.R' \rangle ) ( \iff ) ( \langle A,R \rightarrow \langle A',R' \rangle )</td>
<td></td>
</tr>
<tr>
<td>( \langle X := I, \sigma : S, W \rangle \rightarrow \langle \text{skip}, \langle \sigma : S[X \leq I], W \rangle \rangle )</td>
<td></td>
</tr>
<tr>
<td>( \text{if } B \text{ then } S_1 \text{ else } S_2.R ) ( \rightarrow ) ( \langle \text{if } B' \text{ then } S_1 \text{ else } S_2.R' \rangle ) ( \iff ) ( \langle B,R \rightarrow \langle B',R' \rangle )</td>
<td></td>
</tr>
<tr>
<td>( \text{while } B \text{ S} ) ( \rightarrow ) ( \langle \text{while } B \text{ } \text{S} \text{ } \text{if } \text{true } \text{then } S_1 \text{ else } S_2.R \rangle ) ( \iff ) ( \langle S_1.R \rightarrow \langle S'_1.R' \rangle )</td>
<td></td>
</tr>
</tbody>
</table>

The relation between MSOS and RMSOS definitions assumes that MSOS definitions are in a certain normal form [55] and is made precise by the following theorem, strongly relating MSOS and modular rewriting semantics.
(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. For example, defining halt is done now using only one rule, $C[\text{halt} \ I] \to I$, preserving the desired computational granularity. Additionally, one can also incorporate the configuration as part of the evaluation context, and thus have full access to semantic information on a “by need basis”; the PLT-Redex implementation of context reduction, for example, supports this approach. Notice how the assignment rule, for example, modifies both the redex, transforming it to skip, and the evaluation context, altering the state which can be found at its top. In this framework, constructs like call/cc can be defined with little effort.

In a context reduction semantics of a language, one typically starts by defining the syntax of evaluation contexts. An evaluation context is a program with a “hole”, the hole being a placeholder where the next computational step takes place. If $C$ is such a context and $E$ is some expression whose type fits into the type of the hole of $C$, then $C[E]$ is the program formed by replacing the hole of $C$ by $E$. The characteristic reduction step underlying context reduction is

$$\frac{E \to E'}{C[E] \to C[E']}$$

extending the usual “only-at-the-top” reduction by allowing reduction steps to take place under any desired evaluation context. Therefore, an important part of a context reduction semantics is the definition of evaluation contexts, which is typically done by means of a context-free grammar. The definition of evaluation contexts for our simple language is found in Table 8 (we let $\{\}$ denote the “hole”).

In this BNF definition of evaluation contexts, $S$ is a store variable. Therefore, a “top level” evaluation context will also contain a store in our simple language definition. There are also context-reduction definitions which operate only on syntax (i.e., no additional state is needed), but instead one needs to employ some substitution mechanism (particularly in definitions of $\lambda$-calculus based languages). The rules following the evaluation contexts grammar in Table 8 complete the context reduction semantics of our simple language, which we call CxtRed.

By making the evaluation context explicit and changeable, context reduction is, in our view, a significant improvement over small-step SOS. In particular, one can now define control-intensive statements like halt modularly and at the desired level of computational granularity. Even though the definition in Table 8 gives one the feeling that evaluation contexts and their instantiation come “for free”, the application of the “rewrite in context” rule presented above can be expensive in practice. This is because one needs either to parse/search the entire configuration to put it in the form $C[E]$ for some appropriate

### Table 8

The CxtRed language definition.

| $C\text{Conf}$ | ::= | (CPgm, Store) |
| $CPgm$ | ::= | $\{\}$ | skip | CAExp | CStmt | AExp |
| $C\text{Stmt}$ | ::= | $\{\}$ | CStmt | Stmt | Stmt | X := CAExp | if CBExp then Stmt else Stmt |
| $\text{halt}$ | CAExp | |
| $\text{CBExp}$ | ::= | $\{\}$ | Int | CAExp | CAExp | CBExp | CBExp and BExp | not CBExp |
| $\text{CAExp}$ | ::= | $\{\}$ | Int + CAExp | CAExp | AExp |

$E \to E'$

$C[E] \to C[E']$

| $I_1 + I_2 \to (I_1 + \text{Int} I_2)$ |
| $(P, \sigma)[X] \to (P, \sigma)((\sigma(X))]$ |
| $(P, \sigma)[I + X] \to (P, \sigma[I/X][I])$ when $I = \sigma(X) + 1$ |

| $I_1 \leq I_2 \to (I_1 \leq \text{Int} I_2)$ |
| true and $B \to B$ |
| false and $B \to \text{false}$ |
| not true \to \text{true} |
| not false \to \text{true} |

| if true then $St_1$ else $St_2 \to St_1$ |
| if false then $St_1$ else $St_2 \to St_2$ |
| skip; $St \to St$ |
| $[St] \to St$ |
| $(P, \sigma)[X := I] \to (P, \sigma[I/X][\text{skip}])$ |
| while $B$ $St \to$ if $B$ then $(St; \text{while } B \ St)$ else skip |
| $C[\text{halt} \ I] \to I$ |

$C[\text{skip} \ I] \to \{\}$
C satisfying the grammar of evaluation contexts, or to maintain enough information in some special data-structures to perform the split $C[E]$ using only local information and updates. Moreover, this “matching-modulo-the-CFG-of-evaluation-contexts” step needs to be done at every computation step during the execution of a program, so it may easily become the major bottleneck of an executable engine based on context reduction. Direct implementations of context reduction such as PTL-Redex cannot avoid paying a significant performance penalty [97]. Danvy and Nielsen propose in [27] a technique for efficiently interpreting a restricted form of reduction semantics definitions by means of “refocusing” functions which yield efficient abstract machines. Although these refocusing functions are equationally definable, since we aim here to achieve minimal representational distance, we prefer to translate the definitions into rewriting logic by leaving the rules unchanged and implementing the decompose and plug functions from reduction semantics by means of equations. Next section will present an abstract-machine definition of a programming language in rewriting logic, resembling Felleisen’s CK machine [34], which is obtained by applying Danvy and Nielsen’s technique.

Context reduction is trickier to faithfully capture as a rewrite theory, since rewriting logic, by its locality, always applies the operation $\text{decompose}$ to have an algebraic representation of contexts we extend the signature by adding a constant $C$. View, context-reduction captures context-sensitive rewriting, which, although supported by rewriting logic in the form of a rule in $R$, which is obtained by applying Danvy and Nielsen’s technique.

To faithfully model context-reduction, we make use of two equationally-defined operations: $s2c$, which splits a piece of syntax into a context and a redex, and $c2s$, which plugs a piece of syntax into a context. In our rewriting logic definition, $C[R]$ is not a parsing convention, but rather a constructor conveniently representing the pair (context $C$, redex $R$). In order to have an algebraic representation of contexts we extend the signature by adding a constant $[ ]$, representing the hole, for each syntactic category. The operation $s2c$, presented in Table 9, has an effect similar to what one achieves by parsing in context reduction, in the sense that given a piece of syntax it yields $C[R]$. It is a straightforward, equational definition of the $\text{decompose}$ function used in context-reduction implementations based on the syntax of contexts. We here assume the same restrictions on the context syntax as in [27], namely that the grammar defining them is context-free and that there is always a unique decomposition of an expression into a context and a redex. The operation $c2s$, presented in Table 10, is the equational definition of the $\text{plug}$ function used in interpreting context-reduction definitions, and it is a morphism on the syntax. Notice that (from the defining equations) we have the guarantee that it will be applied only to “well formed”

### Table 9

Equational definition of $s2c$.

<table>
<thead>
<tr>
<th>$s2c((P,S))$</th>
<th>$(C,S)[R]$ if $C[R] = s2c(P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s2c(\text{skip} . J)$</td>
<td>$[[\text{skip} . J]]$</td>
</tr>
<tr>
<td>$s2c(\text{skip} . A)$</td>
<td>$(\text{skip} . C)[R]$ if $C[R] = s2c(A)$</td>
</tr>
<tr>
<td>$s2c(\text{St} . A)$</td>
<td>$(C, A)[R]$ if $C[R] = s2c(\text{St})$</td>
</tr>
<tr>
<td>$s2c(\text{halt} . J)$</td>
<td>$[[\text{halt} . J]]$</td>
</tr>
<tr>
<td>$s2c(\text{halt} . A)$</td>
<td>$(\text{halt} . C)[R]$ if $C[R] = s2c(A)$</td>
</tr>
<tr>
<td>$s2c(\text{while} B \text{ St} )$</td>
<td>$[[\text{while} B \text{ St} ]]$</td>
</tr>
<tr>
<td>$s2c(\text{if } T \text{ then } \text{St} \text{1 else } \text{St} \text{2})$</td>
<td>$[[\text{if } T \text{ then } \text{St} \text{1 else } \text{St} \text{2} ]]$</td>
</tr>
<tr>
<td>$s2c(\text{if } B \text{ then } \text{St} \text{1 else } \text{St} \text{2})$</td>
<td>$(C \text{ then } \text{St} \text{1 else } \text{St} \text{2})[R]$ if $C[R] = s2c(B)$</td>
</tr>
<tr>
<td>$s2c(\text{St})$</td>
<td>$[[\text{St} ]]$</td>
</tr>
<tr>
<td>$s2c(\text{skip} . \text{St} \text{1})$</td>
<td>$[[\text{skip} . \text{St} \text{1} ]]$</td>
</tr>
<tr>
<td>$s2c(\text{St} \text{1} . \text{St} \text{2})$</td>
<td>$(C \text{, St} \text{2})[R]$ if $C[R] = s2c(\text{St} \text{1})$</td>
</tr>
<tr>
<td>$s2c(X::I)$</td>
<td>$[[X::I]]$</td>
</tr>
<tr>
<td>$s2c(\text{X}::\text{A})$</td>
<td>$(\text{X}::\text{C})[R]$ if $C[R] = s2c(\text{X})$</td>
</tr>
<tr>
<td>$s2c(\text{l}::\text{l})$</td>
<td>$[[\text{l}::\text{l}]]$</td>
</tr>
<tr>
<td>$s2c(\text{l}::\text{A})$</td>
<td>$(\text{l}::\text{C})[R]$ if $C[R] = s2c(\text{A})$</td>
</tr>
<tr>
<td>$s2c(\text{A}1::\text{A}2)$</td>
<td>$(\text{A}1::\text{A}2)[R]$ if $C[R] = s2c(\text{A}1)$</td>
</tr>
<tr>
<td>$s2c(\text{T} \text{ and } \text{B}2)$</td>
<td>$[[\text{T} \text{ and } \text{B}2]]$</td>
</tr>
<tr>
<td>$s2c(\text{not } \text{T})$</td>
<td>$[[\text{not } \text{T}]]$</td>
</tr>
<tr>
<td>$s2c(\text{not } \text{B})$</td>
<td>$(\text{not } \text{C})[R]$ if $C[R] = s2c(\text{B})$</td>
</tr>
</tbody>
</table>

| $s2c(X)$ | $[[X]]$ |
| $s2c(X::X)$ | $[[X::X]]$ |
| $s2c(\text{l} + \text{l})$ | $[[\text{l} + \text{l}]]$ |
| $s2c(\text{l} + \text{A})$ | $(\text{l} + \text{C})[R]$ if $C[R] = s2c(\text{A})$ |
| $s2c(\text{A}1 + \text{A}2)$ | $(\text{C} + \text{A}2)[R]$ if $C[R] = s2c(\text{A}1)$ |
The rewrite theory \( \mathcal{R}_{\text{CtxRed}} \) is obtained by adding the rules in Table 11 to the equations of \( s2c \) and \( c2s \).

The \( \mathcal{R}_{\text{CtxRed}} \) definition is a faithful representation of context reduction semantics: indeed, it is easy to see that \( s2c \) recursively finds the redex taking into account the syntactic rules defining a context in the same way a parser would, and in the same way as other current implementations of this technique do it. Also, since parsing issues are abstracted away using equations, the computational granularity is the same, yielding a one-to-one correspondence between the computations performed by the context reduction semantics rules and those performed by the rewriting rules.

**Theorem 2.** Suppose that \( s \succeq \sigma \). Then the following hold:

1. \((p,\sigma)\) parses in \( \text{CtxRed} \) as \((c,\sigma)[r]\) iff \( \mathcal{R}_{\text{CtxRed}} \vdash s2c((p,\sigma)) = (c,\sigma)[r] \);
2. \( \mathcal{R}_{\text{CtxRed}} \vdash \text{c2s}(\text{c}[r]) = \text{c}[r/[]] \) for any valid context \( \text{c} \) and appropriate redex \( r \);
3. \( \text{CtxRed} \vdash \text{c2s}(p,\sigma) \rightarrow (p',\sigma') \) iff \( \mathcal{R}_{\text{CtxRed}} \vdash \cdot((p,\sigma)) \rightarrow (p',\sigma') \) and \( \sigma' \succeq \sigma \);
4. \( \text{CtxRed} \vdash \text{c2s}(p,\sigma) \rightarrow (i) \) iff \( \mathcal{R}_{\text{CtxRed}} \vdash \cdot((p,\sigma)) \rightarrow (i) \);
5. \( \text{CtxRed} \vdash \text{c2s}(p,\bot) \rightarrow^* (i) \) iff \( \mathcal{R}_{\text{CtxRed}} \vdash \text{eval}(p) \rightarrow i \).

**Proof.**

1. By induction on the number of context productions applied to parse the context, which is the same as the length of the derivation of \( \mathcal{R}_{\text{CtxRed}} \vdash s2c(\text{sym}) = \text{c}[r] \), respectively, for each syntactical construct \( \text{sym} \). We only show some of the more interesting cases.

   - **Case \( +x : +x \):** \( +x \) parses as \([[[+x]]] \). Also \( \mathcal{R}_{\text{CtxRed}} \vdash s2c([+x]) = [[+x]] \) in one step (it is an instance of an axiom).

   - **Case \( a_1 \succeq a_2 : a_1 \succeq a_2 \):** \( a_1 \succeq a_2 \) parses as \( a_1 \succeq c[r] \) iff \( a_1 \in \text{Int} \) and \( a_2 \) parses as \( c[r] \) iff \( a_1 \in \text{Int} \) and \( \mathcal{R}_{\text{CtxRed}} \vdash s2c(a_2) = c[r] \) iff \( \mathcal{R}_{\text{CtxRed}} \vdash s2c(a_1 \succeq c)[r] \).

   - **Case \( x:a : x:a \):** \( x:a \) parses as \([[[x:a]]] \) iff \( a \in \text{Int} \), iff \( \mathcal{R}_{\text{CtxRed}} \vdash s2c(x:a) = [[x:a]] \).

   - **Case \( st:a : st:a \):** \( st:a \) parses as \( st \cdot c[r] \) iff \( st = \text{skip} \) and \( a \) parses as \( c[r] \), iff \( st = \text{skip} \) and \( \mathcal{R}_{\text{CtxRed}} \vdash s2c(a) = c[r] \) iff \( \mathcal{R}_{\text{CtxRed}} \vdash s2c(st \cdot a) = st \cdot c[r] \).

   - **Case \( (p,\sigma) : (p,\sigma) \):** \( (p,\sigma) \) parses as \( c[r] \) iff \( p \) parses as \( c'[r] \) and \( c = (c',s) \) iff \( \cdot((p,\sigma)) \rightarrow (c'[r]) \).

---

### Table 10

**Equational definition of \( c2s \).**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c2s([H]) = H )</td>
<td>( c2s([H]) = H )</td>
</tr>
<tr>
<td>( c2s([P,S]) = (c2s(P),S) )</td>
<td>( c2s([P,S]) = (c2s(P),S) )</td>
</tr>
<tr>
<td>( c2s([f]) = [f] )</td>
<td>( c2s([f]) = [f] )</td>
</tr>
<tr>
<td>( c2s(E_1,E_2) = c2s(E_1).c2s(E_2) )</td>
<td>( c2s(E_1,E_2) = c2s(E_1).c2s(E_2) )</td>
</tr>
<tr>
<td>( c2s(\text{halt} E[H]) = \text{halt} c2s(E[H]) )</td>
<td>( c2s(\text{halt} E[H]) = \text{halt} c2s(E[H]) )</td>
</tr>
<tr>
<td>( c2s(\text{while} E_1 E_2[H]) = \text{while} c2s(E_1[H]) c2s(E_2[H]) )</td>
<td>( c2s(\text{while} E_1 E_2[H]) = \text{while} c2s(E_1[H]) c2s(E_2[H]) )</td>
</tr>
<tr>
<td>( c2s(\text{if} E_1 \text{then} E_2 \text{else} E_3[H]) = \text{if} c2s(E_1[H]) \text{then} c2s(E_2[H]) \text{else} c2s(E_3[H]) )</td>
<td>( c2s(\text{if} E_1 \text{then} E_2 \text{else} E_3[H]) = \text{if} c2s(E_1[H]) \text{then} c2s(E_2[H]) \text{else} c2s(E_3[H]) )</td>
</tr>
<tr>
<td>( c2s(E[H]) = c2s(E[H]) )</td>
<td>( c2s(E[H]) = c2s(E[H]) )</td>
</tr>
<tr>
<td>( c2s(E_1 E_2[H]) = c2s(E_1[H]) c2s(E_2[H]) )</td>
<td>( c2s(E_1 E_2[H]) = c2s(E_1[H]) c2s(E_2[H]) )</td>
</tr>
<tr>
<td>( c2s(\text{true}[H]) = \text{true} )</td>
<td>( c2s(\text{true}[H]) = \text{true} )</td>
</tr>
<tr>
<td>( c2s(\text{false}[H]) = \text{false} )</td>
<td>( c2s(\text{false}[H]) = \text{false} )</td>
</tr>
</tbody>
</table>
(5) This part of the proof follows the same pattern as that for the similar property for \textit{SmallStep} (Proposition 3), using the above properties and replacing \textit{smallstep} by \textit{reduction}.

(4) \text{CxtRed} \vdash (p, \sigma) \rightarrow (i) because of \text{CxtRed} \vdash c(\text{skip} \cdot i) \rightarrow (i) iff

\begin{align*}
\text{CxtRed} \vdash & \langle i, \sigma \rangle \rightarrow \langle i, \sigma \rangle \text{[skip]} \iff \\
\text{CxtRed} \vdash & \langle \langle \llbracket l \rrbracket \rangle, \langle \llbracket s \rrbracket \rangle \rangle \rightarrow \langle \langle \llbracket l \rrbracket \rangle, \langle \llbracket s \rrbracket \rangle \rangle \text{[skip]} \iff \\
\text{CxtRed} \vdash & \text{CxtRed} \vdash \neg \text{true} \rightarrow \neg \text{true}.
\end{align*}
Table 12

Rewriting logic theory \( \mathcal{R}_K \) (continuation-based definition of the language).

| \( aexp(l) = I \) | \( bexp(true) = true \) |
| \( aexp(A_1 + A_2) = (aexp(A_1), aexp(A_2)) \) | \( bexp(false) = false \) |
| \( k(aexp(X) \mid K) store(Store) \rightarrow k(store(X) \mid K) store(Store) \) | \( k((\text{stmt}(\text{skip})) \mid l) \rightarrow k(l) \) |
| \( k(aexp(\text{\ldots}) \mid K) store((X = l) \mid Store) \rightarrow k(s(l) \mid K) store(X = s(l) \mid Store) \) |
| \( k(I, I_2 \mid \ldots \mid K) \rightarrow k(I_1 + \text{Int} \mid I_2 \mid K) \) |

Strengths. Context reduction semantics divides SOS rules into computational rules and rules needed to find the redex; the latter are transformed into grammar rules generating the allowable contexts. This makes definitions more compact. It improves over SOS semantics by allowing the context to be changed by execution rules. It can easily deal with control-intensive features. It is more modular than SOS.

Weaknesses. It still only allows “interleaving semantics” for concurrency. Although context-sensitive rewriting might seem to be easily implementable by rewriting, in fact all current implementations of context reduction work by transforming context grammar definitions into traversal functions, thus being as (in)effective as the small-step implementations (one has to perform an amount of work linear in the size of the program for each computational step). However, one might obtain efficient implementations for restricted forms of context-reduction definitions by applying refocusing techniques [27].

9. A continuation-based semantics

The idea of continuation-based interpreters for programming languages and their relation to abstract machines has been well studied (see, for example, [34]). In this section, we propose a rewriting logic theory based on a structure that provides a first-order representation of continuations in the spirit of Wand [95]; this is the only reason why we call this structure a “continuation”; but notice that it can just as well be regarded as a post-order representation of the abstract syntax tree of the program, so one needs no prior knowledge of continuations [34] in order to understand this section. We will show the equivalence of this theory to the context reduction semantics theory.

Based on the desired order of evaluation, the program is sequentialized by transforming it into a list of tasks to be performed in order. This is done once and for all at the beginning, the benefit being that at any subsequent moment in time we know precisely where the next redex is: at the top of the list of tasks. We call this list of tasks a continuation, but is nothing more than a pure first-order flattening of the program and can be easily introduced without appealing to high-order constructs. For example, \( aexp(A_1 + A_2) = (aexp(A_1), aexp(A_2)) \) precisely encodes the order of evaluation: first \( A_1 \), then \( A_2 \), then add the values. Also, \( \text{stmt}(\text{if } B \text{ then } St_1 \text{ else } St_2) = B \rightarrow \text{if}(\text{stmt}(St_1), \text{stmt}(St_2)) \) says that \( St_1 \) and \( St_2 \) are dependent on the value of \( B \) for their evaluation. The fact that we denote the above relation by equality, although we operationally interpret...
it from left to right, indicates that the two terms are structurally equal (and in fact, they are equal in the initial model of the specification)—at any time during the evaluation one could apply the equations backwards and reconstitute the current state of the program being executed.

The top level configuration is constructed by an operator “\_\_” putting together the store (wrapped by a constructor store) and the continuation (wrapped by \(k\)). Also, syntax is added for the continuation items. Here, the distinction between equations and rules becomes even more obvious: equations are used to prepare the context in which a computation step can be applied, while rewrite rules exactly encode the computation steps semantically, yielding the intended computational granularity. Specifically \(pgm, stmt, bexp, aexp\) are used to flatten the program to a continuation, taking into account the order of evaluation. The continuation is defined as a list of tasks, where the list constructor “\_\_\_\_” is associative, having as identity a constant “nothing”. We also use lists of values and continuations, each having an associative list append constructor “\_\_” with identity “\_\_\_\_\_”.

We use variables \(K\) and \(V\) to denote continuations and values, respectively; also, we use \(kl\) and \(vl\) for lists of continuations and values, respectively. The rewrite theory \(R_K\) specifying the continuation-based definition of our example language is given in Table 12. Lists of expressions are evaluated using the following (equationally defined) mechanism:

\[
k((vl, ke, kel) \_ \_ K) = k(ke \_ \_ (vl, nothing, kel) \_ \_ K)
\]

Because in rewriting engines equations are also executed by rewriting, one would need to split the above rule into two rules:

\[
k((vl, ke, kel) \_ \_ K) = k(ke \_ \_ (vl, nothing, kel) \_ \_ K)
\]

\[
k(v \_ \_ (vl, nothing, kel) \_ \_ K) = k((vl, v, kel) \_ \_ K)
\]

The semantics we obtain here for this simple sequential language is an abstract machine, similar in spirit to the one obtainable by applying CPS transformers on an interpreter as in [74] or that obtained by applying refocusing [27] on the context-reduction definition. One slight difference is that we keep the state and the continuation as distinct entities at the top level, rather than embedding the state as part of the context/continuation structure. In a computational logic framework like rewriting logic where the gap between “implementations” and “specifications” is almost inexistent, this continuation-like style can be used to define languages, not only to efficiently interpret them.

An important benefit of this definitional style is that of gaining locality. Now one needs to specify from the context only what is needed to perform the computation. This indeed gives the possibility of achieving “true concurrency”, since rules which do not act on the same parts of the context can be applied in parallel. In [72], we show how the same technique can be used, with no additional effort, to define concurrent languages; the idea is, as expected, that one continuation structure is generated for each concurrent thread or process. Then rewrite rules can apply “truly concurrently” at the tops of continuations.

Strengths. In continuation-based semantics there is no need to search for a redex anymore, because the redex is always at the top. It is much more efficient than direct implementations of evaluation contexts or small-step SOS. Also, this style greatly reduces the need for conditional rules/equations; conditional rules/equations might involve inherently inefficient reachability analysis to check the conditions and are harder to deal with in parallel environments. An important “strength” specific to the rewriting logic approach is that reductions can now apply wherever they match, in a context-insensitive way. Additionally, continuation-based definitions in the RLS style above are very modular (particularly due to the use of matching modulo associativity and commutativity).

Weaknesses. The program is now hidden in the continuation: one has to either learn to like it like this, or to write a backwards mapping to retrieve programs from continuations\(^4\); to flatten the program into a continuation structure, several new operations (continuation constants) need to be introduced, which “replace” the corresponding original language constructs.

9.1. Relation with context reduction

We next show the equivalence between the continuation-based and the context-reduction rewriting logic definitions. The specification in Table 13 relates the two semantics, showing that at each computational “point” it is possible to extract from our continuation structure the current expression being evaluated. For each syntactical construct \(\text{Syn} \in \{AExp,BExp,Stmt, Pgm\}\), we equationally define two (partial) functions:

- \(k2\text{Syn}\) takes a continuation encoding of \(\text{Syn}\) into \(\text{Syn}\); and
- \(k\text{Syn}\) extracts from the tail of a continuation a \(\text{Syn}\) and returns it together with the remainder prefix continuation.

Together, these two functions can be regarded as a parsing process, where the continuation plays the role of “unparsed” syntax, while \(\text{Syn}\) is the abstract syntax tree, i.e., the “parsed” syntax. The formal definitions of \(k2\text{Syn}\) and \(k\text{Syn}\) are given in Table 13.

\(^4\) However, we regard these as minor syntactic details. After all, the program needs to be transformed into an abstract syntax tree (AST) in any of the previous formalisms. Whether the AST is kept in prefix versus postfix order is somewhat irrelevant.
Table 13
Recovering the abstract syntax trees from continuations.

<table>
<thead>
<tr>
<th>k2Pgm(K) = k2Stmt(K')</th>
<th>A if (K',A) = kAExp(K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k2Stmt(nothing) = skip</td>
<td></td>
</tr>
<tr>
<td>k2Stmt(K) = k2Stmt(K'); St if (K',St) = kStmt(K) \land K' \neq nothing</td>
<td></td>
</tr>
<tr>
<td>k2Stmt(K) = St if (K',St) = kStmt(K) \land K' = nothing</td>
<td></td>
</tr>
<tr>
<td>kStmt(K) \text{\textbackslash write}(X) = \langle K' \ X \arrow A \rangle if (K',A) = kAExp(K)</td>
<td></td>
</tr>
<tr>
<td>kStmt(K) \text{\textbackslash while}(K_1, K_2) = \langle K', \text{if } B \text{ then } St; \text{while } B; St \text{ else skip} \rangle</td>
<td></td>
</tr>
<tr>
<td>if (K',B) = kBExp(K) \land B \land K = kStmt(K_2) \land B \neq B_1</td>
<td></td>
</tr>
<tr>
<td>kStmt(K) \text{\textbackslash if}(K_1,K_2) = \langle K', \text{if } B \text{ then } kStmt(K_1) \text{ else } kStmt(K_2) \rangle</td>
<td></td>
</tr>
<tr>
<td>if (K',B) = kBExp(K)</td>
<td></td>
</tr>
<tr>
<td>kStmt(K) \text{\textbackslash halt} = \langle K', \text{halt } A \rangle if (K',A) = kAExp(K)</td>
<td></td>
</tr>
<tr>
<td>k2AExp(K) = A if (\text{nothing},A) = kAExp(K)</td>
<td></td>
</tr>
<tr>
<td>kAExp(K \text{\textbackslash kv}(K_1,K_2) \rightarrow K') = kAExp(VL,K\text{\textbackslash kv}K,K_2 \rightarrow K')</td>
<td></td>
</tr>
<tr>
<td>kAExp(K \text{\textbackslash aexp}(A)) = \langle K,A \rangle</td>
<td></td>
</tr>
<tr>
<td>kAExp(K \text{\textbackslash j}) = \langle K,J \rangle</td>
<td></td>
</tr>
<tr>
<td>kAExp(K \text{\textbackslash j} K_1,K_2 \rightarrow +) = \langle K,k2AExp(K_1) + k2AExp(K_2) \rangle</td>
<td></td>
</tr>
<tr>
<td>k2BExp(K) = B if (\text{nothing},B) = kBExp(K)</td>
<td></td>
</tr>
<tr>
<td>kBExp(K \text{\textbackslash kv}(K_1,K_2) \rightarrow K') = kBExp(VL,K\text{\textbackslash kv}K,K_2 \rightarrow K')</td>
<td></td>
</tr>
<tr>
<td>kBExp(K \text{\textbackslash j}) = \langle K,T \rangle</td>
<td></td>
</tr>
<tr>
<td>kBExp(K \text{\textbackslash and}(K_2)) = (K_1 \text{ and } k2BExp(K_2)) if (K_1,B_1) = kBExp(K)</td>
<td></td>
</tr>
<tr>
<td>kBExp(K \text{\textbackslash not}) = \langle K', \text{not } B \rangle if (K',B) = kBExp(K)</td>
<td></td>
</tr>
</tbody>
</table>

We will show below that for any step CxtRed takes, $R_K$ performs at most one step to reach the same$^5$ configuration. No steps are performed for skip, or for dissolving a block (because these were dealt with when we transformed the syntax into continuation form), or for dissolving a statement into a skip (there is no need for that when using continuations). Also, no steps will be performed for loop unrolling, because this is not a computational step; it is a straightforward structural equivalence. In fact, note that, because of its incapacity to distinguish between computational steps and structural equivalences, CxtRed does not capture the intended granularity of while: it wastes a computation step for unrolling the loop and one when dissolving the while into skip; neither of these steps has any computational content.

In order to clearly explain the relation between reduction contexts and continuations, we go a step further and define a new rewrite theory $R_K'$ which, besides identifying while with its unrolling, adds to $R_K$ the idea of contexts, holes, and pluggable expressions. More specifically, we add a new constant "[\]" and the following equation, again for each syntactical category Syn:

$$k(\text{syn(Syn)} \rightarrow K') = k(\text{syn(Syn)} \rightarrow \text{syn([])} \rightarrow K')$$

replacing the equation for evaluating lists of expressions, namely,

$$k((VL,Ke, Kel) \rightarrow K) = k(Ke \rightarrow (VL, \text{nothing}, Kel) \rightarrow K)$$

by the following equation which puts in a hole instead of nothing:

$$k((VL,Ke,Kel) \rightarrow K) = k(Ke \rightarrow (VL, \text{syn([])}, Kel) \rightarrow K)$$

The intuition for the first rule is that, as we will next show, for any well-formed continuation (i.e., one obtained from a syntactic entity) having a syntactic entity as its prefix, its corresponding suffix represents a valid context where the prefix syntactic entity can be plugged in. As expected, $R_K'$ does not bring any novelty to $R_K$, that is, for any term $t$ in $R_K$, $\text{Tree}_{R_K}(t)$ is bisimilar to $\text{Tree}_{R_K'}(t)$.

**Proposition 4.** For each arithmetic context $c$ in CxtRed and $r \in AExp$, we have that $R_K' \vdash k(aexp(c[r])) = k(aexp(r) \rightarrow aexp(c)))$. Similarly for any possible combination for $c$ and $r$ among $AExp,BExp,Stmt,Pgm,Cfg$.

$^5$ "same" modulo irrelevant but equivalent syntactic notational conventions.
(Note that r in the proposition above needs not be a redex, but can be any expression of the right syntactical category, i.e., pluggable in the hole.)

Proof.  

\[ \begin{align*}  
    & +x = [] +x: \quad \mathcal{R}_K \vdash k(aexp(+x)) = k(aexp(+x) \bowtie aexp([])) 
    
    & a_1 + a_2 = [] + a_2[a_1]: \quad \mathcal{R}_K \vdash k(aexp(a_1 + a_2)) = k((aexp(a_1) \bowtie aexp(a_2)) \bowtie +) 
    
    & = k(aexp(a_1) \bowtie (aexp([a_2]) \bowtie +)) = k(aexp(a_1) \bowtie aexp([a_2])) 
    
    & i_1 + a_2 = i_1 + ([a_2]: \quad \mathcal{R}_K \vdash k(aexp(i_1 + a_2)) = k((aexp(i_1) \bowtie aexp(a_2)) \bowtie +) 
    
    & = k(aexp(a_2) \bowtie (i_1, aexp([])) \bowtie +) = k(aexp(a_2) \bowtie aexp(i_1 + [])). 
    
    & b_1 \text{ and } b_2 = [] \text{ and } b_2[b_1]: \quad \mathcal{R}_K \vdash k(bexp(b_1 + a_2)) = k(bexp(b_1) \bowtie aexp([a_2])) = k(bexp(b_1) \bowtie bexp([a_2] \bowtie +)). 
    
    & t \text{ and } b_2 = [] [t \text{ and } b_2]: \quad \mathcal{R}_K \vdash k(bexp(t \text{ and } b_2)) = k(bexp(t \bowtie b_2) \bowtie aexp([])). 
    
    & \text{skip}\cdot p = \text{skip} \cdot p[a]: \quad \mathcal{R}_K \vdash k(pgm(\text{skip} \cdot a)) = k(\text{stmt}(\text{skip}) \bowtie aexp(a)) 
    
    & = k(aexp(a) \bowtie aexp(a) \bowtie +) 
    
    & = k(aexp(a) \bowtie \text{stmt}(\text{skip}) \bowtie aexp(a)) = k(aexp(a) \bowtie \text{pgm}(\text{skip} \cdot []). 
    
    & \text{All other constructs are dealt with in a similar manner.} \quad \Box 
    
\end{align*} \]

Lemma 1. \( \mathcal{R}_K \vdash k(k_1) = k(k_2) \) implies that for any \( k_{\text{rest}} \), \( \mathcal{R}_K \vdash k(k_1 \bowtie k_{\text{rest}}) = k(k_2 \bowtie k_{\text{rest}}) \).

Proof. We can replay all steps in the first proof, for the second proof, since all equations only modify the head of a continuation. \( \Box \)

By structural induction on the equational definitions, thanks to the one-to-one correspondence of rewriting rules, we obtain the following result:

Theorem 3. Suppose \( s \simeq \sigma \).

1. If \( \text{CxtRed} \vdash (p, \sigma) \rightarrow (p', \sigma') \) then \( \mathcal{R}_K \vdash k(pgm(p)) \bowtie \text{store}(s) \rightarrow^{s} k(pgm(p')) \bowtie \text{store}(s') \) and \( s' \simeq \sigma' \), where \( \rightarrow^{s} \equiv \rightarrow^{0} \cup \rightarrow^{1} \).

2. If \( \mathcal{R}_K \vdash k(pgm(p)) \bowtie \text{store}(s) \rightarrow k(k') \bowtie \text{store}(s') \) then there exists \( p' \) and \( \sigma' \) such that \( \text{CxtRed} \vdash (p, \sigma) \rightarrow^{*} (p', \sigma'), \mathcal{R}_K \vdash k(pgm(p')) = k(k') \) and \( s' \simeq \sigma' \).

3. \( \text{CxtRed} \vdash (p, \bot) \rightarrow^{*} i \) iff \( \mathcal{R}_K \vdash (p) \rightarrow i \) for any \( p \in \text{Pgm} \) and \( i \in \text{Int} \).

Proof (Sketch).

1. First, one needs to notice that rules in \( \mathcal{R}_K \) correspond exactly to those in \( \text{CxtRed} \). For example, for \( i_1 + i_2 \rightarrow i_1 +_{\text{Int}} i_2 \), which can be read as \( \langle c, \sigma \rangle[i_1 + i_2] \rightarrow \langle c, \sigma \rangle[i_1 +_{\text{Int}} i_2] \) we have the rule \( k((i_1 + i_2) \bowtie +) \rightarrow k(i_1 +_{\text{Int}} i_2) \bowtie k_{\text{rest}} \) which, taking into account the above results, has, as a particular instance: \( k(pgm(c[i_1 + i_2])) \rightarrow k(pgm(c[i_1 +_{\text{Int}} i_2])) \). For \( \langle c, \sigma \rangle[x := i] \rightarrow \langle c, \sigma[i/x] \rangle \bowtie \text{skip} \) we have \( k(i \bowtie \text{write}(x) \bowtie k) \bowtie \text{store}(s) \rightarrow k(k) \bowtie \text{store}(s[x := i]) \) which again has as an instance: \( k(pgm(c[x := i])) \bowtie \text{store}(s) \rightarrow k(c[i/x] \bowtie \text{store}(s[x := i]) \).

2. Actually \( \sigma' \) is uniquely determined by \( s' \) and \( p' \) is the program obtained by advancing \( p \) all non-computational steps—which were dissolved by \( \text{pgm} \), or are equationally equivalent in \( \mathcal{R}_K \), such as unrolling the loops—, then performing the step similar to that in \( \mathcal{R}_K \).

3. Using the previous two statements, and the rules for halt or end of the program from both definitions. We exemplify only halt, the end of the program is similar, but simpler. For \( \langle c, \sigma \rangle[\text{halt} := i] \rightarrow i \) we have \( k(i \bowtie \text{halt} \bowtie k) \rightarrow k(i) \), and combined with \( \mathcal{R}_K \vdash \text{result}(k(i) \bowtie \text{store}(s)) = i \) we obtain \( \mathcal{R}_K \vdash \text{result}(k(pgm(c[\text{halt} := i])) \bowtie \text{store}(s)) \rightarrow i. \quad \Box \)

10. The chemical abstract machine

Berry and Boudol’s chemical abstract machine, or Cham \[8\], is both a model of concurrency and a specific style of giving operational semantics definitions. Properly speaking, it is not an SOS definitional style. Berry and Boudol identify a number of limitations inherent in SOS, particularly its lack of true concurrency, and what might be called SOS’s rigidity and slavery to syntax \[8\]. They then present the Cham as an alternative to SOS. In fact, as already pointed out in \[50\], what the Cham is,
is a particular definitional style within RLS. That is, every Cham is, by definition, a specific kind of rewrite theory; and Cham computation is precisely concurrent rewriting computation; that is, proof in rewriting logic.

The basic metaphor giving its name to the Cham is inspired by Banâtre and Le Métayer’s GAMMA language [6]. It 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. It is possible to define a variety of chemical abstract machines. Each of them corresponds to a rewrite theory satisfying certain common conditions.

There is a common syntax shared by all chemical abstract machines, with each machine possibly extending the basic syntax by additional function symbols. The common syntax is typed, and can be expressed as the following order-sorted signature Ω:

- `sorts Molecule, Molecules, Solution`
- `subsorts Solution < Molecule < Molecules`
- `op λ :→ Molecules`
- `op _._ : Molecules Molecules→Molecules`
- `op [_] : Molecules→Solution. *** membrane operator`
- `op _⇒_ : Molecule Solution→Molecule. *** airlock operator`

A Cham is then a rewrite theory \( G = (\Sigma, AC, R) \), with \( \Sigma \supseteq \Omega \), together with a partition \( R = \text{Reaction} \cup \text{Heating} \cup \text{Cooling} \cup \text{AirlockAx} \). The associativity and commutativity (AC) axioms are asserted of the operator \( _.._ \), which has identity \( λ \). The rules in \( R \) may involve variables, but are subject to certain syntactic restrictions that guarantee an efficient form of AC matching [8]. `AirlockAx` is the bidirectional rule\(^6\) \( \{m,M\} \leftarrow \{m \to [M]\} \), where \( m \) is a variable of sort `Molecule` and \( M \) a variable of sort `Molecules`. The purpose of this axiom is to choose one of the molecules \( m \) in a solution as a candidate for reaction with other molecules outside its membrane. The Heating and Cooling rules can typically be paired, with each rule \( t \rightarrow t' \in \text{Heating} \) having a symmetric rule \( t' \rightarrow t \in \text{Cooling} \), and vice-versa, so that we can view them as a single set of bidirectional rules \( t' \leftrightarrow t \in \text{Heating} \cup \text{Cooling} \).

Berry and Boudol [8] make a distinction between rules, which are rewrite rules specific to each Cham—and consist of the Reaction, Heating, and Cooling rules—and laws which are general properties applying to all Chams for governing the admissible transitions. The first three laws, the Reaction, Chemical and Membrane laws, just say that the Cham evolves by AC-rewriting. The fourth law states the axiom `AirlockAx`. The Reaction rules are the heart of the Cham and properly correspond to state transitions. The rules in Heating–Cooling express structural equivalence, so that the Reaction rules may apply after the appropriate structurally equivalent syntactic form is found. A certain strategy is typically given to address the problem of finding the right structural form, for example to perform “heating” as much as possible. In rewriting logic terms, a more abstract alternative view is to regard each Cham as a rewrite theory \( G = (\Sigma, AC \cup \text{Heating} \cup \text{Cooling} \cup \text{AirlockAx}; \text{Reaction}) \), in which the Heating–Cooling rules and the AirlockAx axiom have been made part of the theory’s equational axioms. That is, we can more abstractly view the Reaction rules as applied modulo ACI \( \cup \text{Heating} \cup \text{Cooling} \cup \text{AirlockAx} \).

As Berry and Boudol demonstrate in [8], the Cham is particularly well suited to give semantics to concurrent calculi, yielding considerably simpler definitions than those afforded by SOS. In particular, [8] presents semantic definitions for the TCCS variant of CCS, a concurrent \( \lambda \)-calculus, and Milner’s \( \pi \)-calculus. Milner himself also used Cham ideas to provide a compact formulation of his \( \pi \)-calculus [59]. Since our example language is sequential, it cannot take full advantage of the Cham’s true concurrent capabilities. Nevertheless, there are interesting Cham features that, as we explain below, turn out to be useful even in this sequential language application. A Cham semantics for our language is given in Table 14. Note that, since the Cham is itself a rewrite theory, in this case there is no need for a representation in RLS, nor for a proof of correctness of such a representation; that is, the “representational distance” in this case is equal to 0. Again, RLS does not advocate any particular definitional style: the Cham style is just one possibility among many, having its own advantages and limitations.

The Cham definition for our simple programming language takes the CxtRed definition in Table 8 as a starting point. More precisely, we follow the “refocusing” technique [27]. We distinguish two kinds of molecules: syntactic molecules and store molecules. Syntactic molecules are either language constructs or evaluation contexts and we will use \( [\_ | \_] \) as a molecule constructor for stacking molecules. We let \( C \) range over syntactic molecules representing stacked contexts. Store molecules are pairs \( (x,i) \), where \( x \) is a variable and \( i \) is an integer. The store is a solution containing store molecules. Then the definition of “refocusing” functions is translated into heating/cooling rules, bringing the redex to the top of the syntactic molecule. This allows for the reduction rules to only operate at the top, in a conceptually identical way as for continuation based definitions in Table 12, both of them following the same methodology introduced in [72].

One can notice a strong relation between our Cham and CxtRed definitions, in the sense that a step performed using reduction under evaluation contexts is equivalent to a suite of heating steps followed by one transition step and then by as many cooling steps as possible. That is, given programs \( P, P' \) and states \( \sigma, \sigma' \):

\[
\text{CxtRed} \vdash (P,\sigma) \rightarrow (P',\sigma') \iff \text{CHAM} \vdash P,[[\sigma]] \rightarrow^*; \rightarrow^1; \leftarrow^* P',[[\sigma']] \]

\(^6\) Which is of course understood as a pair of rules, one in each direction.
Moreover, one can notice that airlock rules were introduced to select elements from a set without specifying the rest of the members. The CHAM language definition allows for skip, skip, and the introduction of new variables. Table 14 illustrates the CHAM language definition:

| {|
| {|
| molecules) | skip
| skip | {|
| X:A | C := [A | X:=[] | C]]
| [S; S2 | C := [S1 | []; S2 | C]]
| [if B then S1 else S2 | C := [B | [if [] then S1 else S2 | C]]
| [halt A | C := [A | [halt [] | C]]
| [A1 <= A2 | C := [A1 | [] < A2 | C]]
| [| C := [A | [] <= [] | C]]
| [halt B | C := [B | [] <= [] | C]]
| [A1 < A2 | C := [A1 | [] <= [] | A2 | C]]
| [I + A | C := [A | [I + [] | C]]

| l1 + l2 → (l1 +int l2)
| [X | C]>({XJ} + σ) → [I | C]>({XJ} + σ)
| (+X | C]>({XJ} + σ) → [I + int 1 | C]>({XJ} + int 1) + σ]

| l1 <= l2 → (l1 <= int l2)
| true and B → B
| false and B → false
| not true → false
| not false → true

| if true then S1 else S2 → S1
| if false then S1 else S2 → S2
| skip; S1 → S1
| [S; S1 → S1
| [X:=I | C]>({XJ} + σ) → [skip; C]>({XJ} + σ)
| [while B S1 | C] → [if B then (S1; while B S1) else skip | C]
| [halt I | C] + σ → I

Note that we could not use the existing airlock mechanism to stack evaluation contexts since that could lead to unsound computations. Indeed, say one would use constructs {..,..} to stack contexts, replacing the {...} construct. Then by applying heating on skip: 3/4/5, one can obtain the following sequence (of structurally equivalent molecules)

```
skip; 5/(2/x) → [5/(2/x) + [skip; []]] → [5/(2/x) + [skip; []]]
```

Now, by applying the cooling, then heating rules for airlock, one obtains the following sequence (of, again, equivalent molecules)

```
[5/(2/x) + [skip; []] + [5/(2/x) + [skip; []]])
```

Finally, by applying cooling rules for contexts, we obtain the sequence:

```
[5/(2/x) + [skip; []] + [5/(2/x) + [skip; []]])
```

However, skip: 5/(2/x) and skip: 2/(5/x) are obviously not structurally equivalent.

The above language definition does not exhibit the strengths of the Cham, since Cham was designed to handle easily concurrent constructs, which are missing from our language. However, making the above language concurrent in Cham comes at no additional effort. One can execute multiple programs at the same time, sharing the store, simply by putting them together, and together with the store at the top-level solution and replacing the rule for the end of the program by skip. I → I, to allow all programs to finish their evaluation and keep the results.

When Cham definitions follow the style in Table 14, 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 is transformed into:

\[ X := l \mid C \vdash \langle X, l \rangle, \sigma \] → [skip] \[ C \vdash \langle X, l \rangle, \sigma \]

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 final state anymore.

**Strengths.** Being a special case of rewriting logic, it inherits many of the benefits of rewriting 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 final states, possibly leading to a combinatorial explosion. Rewriting strategies such as those in \([10,93,29]\) 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 state of the machine is simple, 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 call/cc. Even though it was easy to define halt because we simply discarded the entire solution, it would seem hard or impossible to define more complex control-intensive language features in Cham.

11. Experiments

RLS definitions, being executable, actually are also interpreters for the programming languages they define. One can take an RLS executable definition as is and execute it on top of a rewrite engine.

However, one should not wrongly conclude from this that in order to make any use of RLS definitions of programming languages, in particular of those following the various definitional styles proposed in this paper, one must have an advanced rewrite engine. In fact, one can implement interpreters for languages given an RLS definition using one’s programming language of choice. Although the proposed RLS definitions follow the same style and intuitions, and have the same strengths and limitations as their original formulation in their corresponding definitional styles, we believe that automating the process of generating interpreters from the rewriting logic language definitions following a specific operational semantics style should be easier than doing it directly from the original definition, since the rewriting logic definition is already executable. Furthermore, since most of the definitional styles presented in this paper use a restricted form of rewriting, one can hope for automatic translations of those definitions into interpreters in programming languages offering a limited support for matching and rewriting. To test this claim, we have manually but mechanically translated the RLS definitions for all styles (except for MSOS and the Cham) in Haskell, Ocaml, and Prolog. Appendix 13 discusses our translation procedures into these programming languages.

We compare the running times and memory requirements of the interpreters derived mechanically using the above-mentioned procedures, with those of the “free” interpreters given by executing the definition “as-is” on two rewrite engines (marked with \(\ddagger\) in the tables), namely \texttt{Asf+Sdf 1.5} (a compiler) and \texttt{Maude 2.2} (a fast interpreter with good tool support), as well as with those obtained executing off-the-shelf interpreter implementations in Scheme, used in teaching programming languages (marked with \(\ddagger\ddagger\) in the tables). For Scheme we have used PLT-Scheme as an interpreter and language interpreter implementations from [35], chapters 3.9 (evaluation semantics) and 7.3 (continuation based semantics), and a PLT-Redex definition given as example in the installation package (for context reduction). Big-step interpreters are also compared against bc, a C-written interpreter for a subset of C working only with integers (bc comes as part of UNIX; type “man bc” for documentation), and two interpreters implemented using monads in Haskell and Ocaml (we mark these interpreters with \(\triangleright\) in Table 16). Since RLS definitions of MSOS and Cham definitions rely intensively on matching modulo associativity and commutativity, which is only supported by Maude, we have only performed some experiments on their RLS definitions in Maude. For Cham we preferred to give the times obtained by using the novel transformations and strategies presented in Section 10 for making the specification “more executable”. Using the specification as is, Cham is extremely ineffective when executed: it takes about 1205MB of memory and 188 s to search for the solution of running the Collatz program (explained below) up to 3.

One may naturally ask: “What is the point of all these experiments? They show little or nothing to support the RLS resulting definitions compared to their original definitions, and only show what programs (interpreters) in what programming languages are more efficient than others.” Our goal here is to convey the reader our strong belief, supported by empirical evaluation, that the working language designer may be better off in practice formally defining a desired language, using some preferred definitional style, than implementing an interpreter in an ad-hoc way for that language, even in a preferred
Table 15
Programs used in evaluation: (a) a tower of loops, each performing two iterations; (b) a program testing Collatz’s conjecture up to 300.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
</table>
| \#steps := \#0;  
\#n := \#300;  
while (not \(#n=2\))  
\#n := \#n - 1;  
\#x0 := \#0;  
while (++\#x0<=2)  
\#x1 := \#0;  
while (++\#x1<=2)  
...  
\#x18 := \#0;  
while (++\#x18<=2)  
  \#skip;  
...  
  \#if (\#r=0)  
    \#n := \#q  
  \#else  
    \#n := 3*\#n + 1  
  \#end  
\#end |


... after a finite number of steps, the value of \(n\) will become 1. To make the program more computation-intensive (and also to maximize the number of language constructs used), we here use repeated subtraction to compute division. We also count in steps the cumulative number of operations performed until 1 is reached for all numbers tested and return it as the result of the program. The source code for the programs used is presented in Table 15.

Tables 16, 17, 18, and 19, give for each definitional style the running time of the various interpreters. For the largest number \(n\) (18) of nested loops, peak memory usage was also recorded. Times are expressed in seconds. A limit of 700 mb was set on memory usage, to avoid swapping; the symbol “-” found in a table cell signifies that the memory limit was reached.

Prolog yields pretty fast interpreters. However, for backtracking reasons, it needs to maintain the stack of all predicates tried on the current path, thus the amount of memory grows with the number of computational steps. The style promoted in [35] seems to also take into account efficiency. Its only drawback is the fact that it looks more like an implementation, the representational distance to the big-step definition being much bigger than in interpreters based on RLS. The PLT-Redex implementation of context reduction seems to serve more a didactic purpose. It compensates for lack of speed by providing a nice interface and the possibility to visually trace a run. The rewriting logic implementations seem to be quite efficient in terms of speed and memory usage, while keeping a minimal representational distance to the operational semantics definitions.

In particular, RLS definitions interpreted in Maude are comparable in terms of efficiency with the interpreters in Scheme, while having the advantage of being formal definitions. The main reason for Maude and Scheme being slower than the others, is because they are both interpreters while the others are compilers. It is well known that compilers usually generate executables one order of magnitude faster than their interpreted versions. Also, it is good to notice that the interpreter...
Table 16
Execution times for big-step definitions.

<table>
<thead>
<tr>
<th>N nested loops(1...2)</th>
<th>N</th>
<th>15</th>
<th>16</th>
<th>18</th>
<th>Memory for 18</th>
<th>Up to 300</th>
<th>Collatz’ conjecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asf+Sdf</td>
<td>1.7</td>
<td>2.9</td>
<td>11.6</td>
<td>13 mb</td>
<td>265.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>0.3</td>
<td>0.6</td>
<td>2.3</td>
<td>&lt;1 mb</td>
<td>13.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haskell</td>
<td>0.3</td>
<td>0.7</td>
<td>2.8</td>
<td>4 mb</td>
<td>32.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haskell (monads)</td>
<td>0.6</td>
<td>1.4</td>
<td>4.4</td>
<td>3 mb</td>
<td>58.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maude</td>
<td>3.8</td>
<td>7.7</td>
<td>31.5</td>
<td>6 mb</td>
<td>184.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocaml</td>
<td>0.5</td>
<td>1.1</td>
<td>5.0</td>
<td>1 mb</td>
<td>10.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocaml (monads)</td>
<td>0.5</td>
<td>0.9</td>
<td>3.8</td>
<td>2 mb</td>
<td>21.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prolog</td>
<td>1.6</td>
<td>1.9</td>
<td>7.6</td>
<td>316 mb</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheme [35]</td>
<td>3.8</td>
<td>7.4</td>
<td>30.2</td>
<td>13 mb</td>
<td>122.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 17
Execution times for small-step definitions.

<table>
<thead>
<tr>
<th>N nested loops(1...2)</th>
<th>N</th>
<th>15</th>
<th>16</th>
<th>18</th>
<th>Memory for 18</th>
<th>Up to 300</th>
<th>Collatz’ conjecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asf+Sdf</td>
<td>11.9</td>
<td>25.7</td>
<td>115.0</td>
<td>9 mb</td>
<td>769.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haskell</td>
<td>3.2</td>
<td>7.0</td>
<td>31.64</td>
<td>3 mb</td>
<td>167.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maude</td>
<td>63.4</td>
<td>131.2</td>
<td>597.4</td>
<td>6 mb</td>
<td>&gt;1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocaml</td>
<td>1.0</td>
<td>2.2</td>
<td>9.9</td>
<td>1 mb</td>
<td>21.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prolog</td>
<td>7.0</td>
<td>14.5</td>
<td>—</td>
<td>&gt;700 mb</td>
<td>—</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

obtained by mechanically compiling the RLS definition in Ocaml can reach the speed of the hand-optimized, C-written bc interpreter.

12. Related work

There is much related work on frameworks for defining programming languages. Without trying to be exhaustive, we mention some of them. We do not try to give detailed comparisons with each approach, but limit ourselves to making some high-level remarks. Also, we do not discuss any of the approaches, such as SOS, MSOS, context reduction, or the Cham, which we have already discussed in the body of the paper.

12.1. Algebraic denotational semantics

This approach, (see [95,39,14,62] for early papers and [37,88] for two more recent books), is a special case of RLS, namely, the case in which the rewrite theory \( R \) defining language \( L \) is an equational theory. While algebraic semantics shares a number of advantages with RLS, its main limitation is that it is well suited for giving semantics to deterministic languages, but not well suited for concurrent language definitions. At the model-theoretic level, initial algebra semantics, pioneered by Joseph Goguen, is the preferred approach (see, for example, [39,37]), but other approaches, based on loose semantics or on final algebras, are also possible.

12.2. Other RLS work

RLS is a collective international project. Through the efforts of various researchers, there is by now a substantial body of work demonstrating the usefulness of this approach [3–5,12,13,17,18,21,24,26,30–33,36,42,43,45,53,56,73,75,76,84–86,89–91]. A first snapshot of the RLS project was given in [56], and a second in [57]. This paper can be viewed as third snapshot focusing on the variety of definitional styles supported. In particular, a substantial body of experience in giving programming language definitions, and using those definitions both for execution and for analysis purposes has already been gathered. For example, Java 1.4 (see also [20] for a complete formal semantics) and the JVM (see [33,30]) have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [33,31]. A semantics of a Caml-like language with threads was discussed in detail in [56], and a modular rewriting logic semantics of a subset of CML has been given in [18] using the Maude MSOS tool [19]. A definition of the Scheme language has been given in [26]. Other language case studies, all specified
Table 18
Execution times for context reduction definitions.

<table>
<thead>
<tr>
<th>N nested loops(1…2)</th>
<th>Collatz’ conjecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>9</td>
</tr>
<tr>
<td>AtS*StD</td>
<td>0.6</td>
</tr>
<tr>
<td>Haskell</td>
<td>0.1</td>
</tr>
<tr>
<td>Maude</td>
<td>0.8</td>
</tr>
<tr>
<td>Ocaml</td>
<td>0.0</td>
</tr>
<tr>
<td>Prolog</td>
<td>0.1</td>
</tr>
<tr>
<td>Scheme: PLT-Redex</td>
<td>198.2</td>
</tr>
</tbody>
</table>

Table 19
Execution times for continuation-based definitions.

<table>
<thead>
<tr>
<th>N nested loops(1…2)</th>
<th>Collatz’ conjecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>15</td>
</tr>
<tr>
<td>AtS*StD</td>
<td>2.5</td>
</tr>
<tr>
<td>Haskell</td>
<td>0.6</td>
</tr>
<tr>
<td>Maude</td>
<td>8.4</td>
</tr>
<tr>
<td>Ocaml</td>
<td>0.5</td>
</tr>
<tr>
<td>Prolog</td>
<td>3.0</td>
</tr>
<tr>
<td>Scheme [35]</td>
<td>5.9</td>
</tr>
</tbody>
</table>

in Maude, include BC [13], CCS [90,13], CIAO [85], Creol [43], ELOTOS [89], MSR [16,83], PLAN [84,85], the ABEL hardware description language [45], SILF [42], FUN [72], Orc [4,5], and the π-calculus [86].

12.3. Higher-order approaches

The most classic higher-order approach, although not exactly operational, is denotational semantics [78,79,77,63]. Denotational semantics has some similarities with its first-order algebraic cousin mentioned above, since both are based on semantic equations. Two differences are: (i) the use of first-order equations in the algebraic case versus the higher-order ones in traditional denotational semantics; and (ii) the kinds of models used in each case. A related class of higher-order approaches uses higher-order functional languages or higher-order theorem provers to give operational semantics to programming languages. Without trying to be comprehensive, we can mention, for example, the use of Scheme in [35], the use of ML in [69], and the use of Common Lisp within the ACL2 prover in [46]. There is also a body of work on using monads [61,94,47] to implement language interpreters in higher-order functional languages; the monadic approach has better modularity characteristics than standard SOS. A third class of higher-order approaches are based on the use of higher-order abstract syntax (HOAS) [68,41] and higher-order logical frameworks, such as LF [41] or λ-Prolog [66], to encode programming languages as formal logical systems. For a good example of recent work in this direction see [58] and references there.

12.4. Logic-programming-based approaches

Going back to the Centaur project [11,25], logic programming has been used as a framework for SOS language definitions. Note that λ-Prolog [66] belongs both in this category and in the higher-order one. For a recent textbook giving logic-programming-based language definitions, see [80].

12.5. Abstract state machines

Abstract State Machine (ASM) [40] can encode any computation and have a rigorous semantics, so any programming language can be defined as an ASM and thus implicitly be given a semantics. Both big- and small-step ASM semantics have been investigated. The semantics of various programming languages, including, for example, Java [81], has been given using ASMs. There are interesting connections between ASMs and rewriting logic, but their discussion is beyond the scope of this paper.

13. Conclusions

In this paper, we have tried to show how RLS can be used as a logical framework for operational semantics definitions of programming languages. In particular, by showing in detail how it can faithfully capture big-step and small-step SOS,
Table 20  Execution times for RLS definitions interpreted in Maude.

<table>
<thead>
<tr>
<th>N nested loops(1…2)</th>
<th>15</th>
<th>16</th>
<th>18</th>
<th>Memory for 18</th>
<th>Collatz’ conjecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-step</td>
<td>3.8</td>
<td>7.7</td>
<td>31.5</td>
<td>6 mb</td>
<td>184.5</td>
</tr>
<tr>
<td>Small-step</td>
<td>63.4</td>
<td>131.2</td>
<td>597.4</td>
<td>6 mb</td>
<td>1249.1</td>
</tr>
<tr>
<td>Context-reduction</td>
<td>76.2</td>
<td>162.8</td>
<td>713.2</td>
<td>6 mb</td>
<td>1931.6</td>
</tr>
<tr>
<td>Continuation-based</td>
<td>8.4</td>
<td>15.6</td>
<td>63.2</td>
<td>7 mb</td>
<td>483.9</td>
</tr>
<tr>
<td>MSOS</td>
<td>61.9</td>
<td>127.4</td>
<td>566.3</td>
<td>6 mb</td>
<td>1421.5</td>
</tr>
<tr>
<td>Cham</td>
<td>15.7</td>
<td>31.5</td>
<td>129.2</td>
<td>6 mb</td>
<td>618.0</td>
</tr>
</tbody>
</table>

MSOS, context reduction, continuation-based semantics, and the Cham, we hope to have illustrated what might be called its *ecumenical* character; that is, its flexible support for a wide range of definitional styles, without forcing or pre-imposing any given style. In fact, we think that this flexibility makes RLS useful as a way of exploring new definitional styles. For example, our discussion on the Cham makes clear that the Cham proponents are dissatisfied with the lack of true concurrency in standard SOS. For highly concurrent languages, such as mobile languages, or for languages involving concurrency, real-time and/or probabilities, it seems clear to us that a centralized approach forcing an interleaving semantics becomes increasingly unnatural. We have, of course, refrained from putting forward any specific suggestions in this regard: that was not the point of an ecumenical paper. But we think that new definitional styles are worth investigating; and hope that RLS in general, and this paper in particular, will stimulate such investigations.

**Acknowledgments**

We thank our fellow researchers in the RLS project, including Wolfgang Ahrendt, Musab Al-Turki, Marcelo d’Amorim, Eyvind W. Axelsen, Cristiano Braga, Illiano Cervesato, Fabricio Chalub, Feng Chen, Manuel Clavel, Azadeh Farzan, Alejandra Garrido, Mark Hills, Einar Broch Johnsen, Ralph Johnson, Michael Katelman, Laurentiu Leustean, Narciso Martí-Oliet, Olaf Owe, Stefan Reich, Andreas Roth, Juan Santa-Cruz, Ralf Sasse, Koushik Sen, Mark-Oliver Stehr, Carolyn Talcott, Prasanna Thati, Ram Prasad Venkatesan, and Alberto Verdejo, for their many contributions, which have both advanced the project and stimulated our ideas. We also thank the students at UIUC who attended courses on programming language design, semantics, and formal methods, who provided important feedback and suggestions. Last but not least, we thank Mark Hills for fruitful discussions and his help in adjusting the PLT-Redex implementation to suit our needs.

**Appendix A. Obtaining interpreters from RLS definitions**

Since the definitions presented above are deterministic and use a restricted form of rewriting (with the exception of MSOS and CHAM), we believe it is straightforward to generate interpreters from them in languages having built-in support for pattern matching and abstract data types. The main principle we use is to translate rewriting rules into evaluation functions. Since the store was defined separately and relies on matching modulo associativity and commutativity, we abstract it away, assuming each such language comes with a pre-defined store.

In the following we will show, with the definitions of assignment from big-step and continuation semantics how their translation appears as part of the chosen implementation languages. Since functional languages have a particular way of declaring abstract data types, you will notice that the syntax of the program looks different in different languages. However, assuming the existence of an external parser, we could ask from that parser to give as output terms of the abstract data type in the corresponding language.

### A.1. Big-Step based definitions

The rewriting rule for assignment in big-step is

\[
\langle X := A, S \rangle \rightarrow \langle S'[X \approx I] \rangle \quad \text{if} \quad (A, S) \rightarrow (I, S')
\]

ASF+SDF Since ASF+SDF is a rewriting engine, translating RLS specifications to ASF+SDF interpreters is mostly a matter of using a different notation. In fact ASF+SDF adopts a notation with setting the premises above the line, close to the original semantics.

\[
\text{[] <I,S1> := <A,S>}
\]

\[
\text{---------------------------}
\]

\[
\text{<X := A,S> = bind(S1,X,I)}
\]
Haskell: We use \( \text{Scfg}(st,s) \) and \( \text{Acfg}(a,s) \), etc., to encode configurations \( \langle st,s \rangle \) and \( \langle a,s \rangle \), respectively. We define an evaluation function for each type of configuration, for example \( e\text{Stmt} \) is the function evaluating \( \text{Scfg} \) configurations and \( e\text{AExp} \) is evaluating \( \text{Acfg} \) configurations. The matching of the evaluation of premises is performed by using the let construct.

\[
e\text{Stmt} (\text{Scfg} (\text{Assign} x a) s) = \\
\quad \text{let} (\text{Acfg} (\text{Int} i) s1) = e\text{AExp} (\text{Acfg} a s) \\
\quad \text{in} (\text{bind} s1 x i)
\]

Maude: Since Maude is the standard execution engine for rewriting logic specifications, the rules here are the ones in the specification.

\[
\text{r1} \langle X := A, S \rangle \rightarrow S1[X <- I] \text{ if } \langle A, S \rangle \rightarrow \{I, S1\}.
\]

Ocaml: Since Ocaml supports polymorphic functions, we only need to define one evaluation function for all constructs. Then matching is used to obtain the starting term and \text{match} \ldots \text{with} \ldots \text{is used for evaluating the premises.}

\[
\text{let rec eval} = \text{function} \\
\quad \ldots \\
\quad | \text{Scfg}(\text{Assign}(x,a),s) \rightarrow \\
\quad \quad (\text{match eval} (\text{Acfg}(a,s)) \text{ with} \\
\quad \quad \quad \text{Acfg}(\text{Int}(i),s1) \rightarrow \\
\quad \quad \quad \text{(bind } s1 x i))
\]

Prolog: In Prolog we define a relation for each type of configuration and use unification for matching only purposes. Note that while in Ocaml, constructors of abstract data types start with capital letter, in Prolog this would correspond to variables, so we need to use \text{scfg}, \text{acfg}, etc., to encode configurations.

\[
\text{eStmt}(\text{scfg}(X = A, S), S2) :- e\text{AExp}(\text{acfg}(AE,S),\text{acfg}(I,S1)), \\
\quad \text{bind}(S1,X,I,S2).
\]

A.2. Continuation-based definitions

Recall that the RLS semantics for assignment consists of an equation and a rule:

\[
\text{stmt}(X := A) = \text{aexp}(A) \rightarrow \text{write}(X) \\
\text{kl}(\text{write}(X) \rightarrow K) \text{ store}(\text{Store}) \rightarrow k(K) \text{ store}(\text{Store}[X <- I])
\]

\(\text{Asf+Sdf} \) Again, the translation to \(\text{Asf+Sdf} \) implies minimal or no modifications. Note that \(\text{Asf+Sdf} \) makes no distinction between equations and rules, all of them being written as equations.

\[
\begin{align*}
\# \text{stmt}(X := A) &= \text{aexp}(A) \rightarrow \text{write}(X) \\
\# \text{kl}(\text{int}(I) \rightarrow \text{write}(X) \rightarrow K) \text{ store}(\text{Store}) &= k(K) \text{ store}(\text{bind}(\text{Store},X,I))
\end{align*}
\]

Haskell: The continuation concatenation is replaced by list concatenation. The evaluation rules are transformed into a recursive evaluation function acting at the top of the state.

\[
\text{stmt (Assign x a)) = (aexp a) @ [\text{Write } x] \\
\text{result (Kval (Vint i):Kwrite x:k) s = result k (bind s x i)}
\]

Maude: Representation in Maude is the exact rewriting logic definition.

\[
\begin{align*}
\text{eq stmt}(X := A) &= \text{aexp}(A) \rightarrow \text{write}(X) \\
\text{rl k}(\text{int}(I) \rightarrow \text{write}(X) \rightarrow K) \text{ store}(\text{Store}) &= k(K) \text{ store}(\text{Store}[X <- I])
\end{align*}
\]

Ocaml: A similar approach as that for Haskell.

\[
\text{let rec stmt = function} \\
\quad \ldots \\
\quad | \text{Assign}(x, a) \rightarrow (\text{aexp a}) @ [\text{Write } x] \\
\text{let rec result s = function} \\
\quad \ldots \\
\quad | (\text{Kval (Vint i):Kwrite x:k}) \rightarrow \text{result (bind s x i)} k
\]
Prolog: Same approach as for the functional languages above, but we now define (functional) evaluation relations for functions decomposing the program and a one-step rewrite relation for the top-level evaluation process.

\[
\text{step}(\text{conf}(S), v([I]), k([\text{write}(X)|K]))) =
\text{conf}(\text{store}(S), v([V]), k(K))
\]

References


