Program Verification: Lecture 23

José Meseguer

University of Illinois
at Urbana-Champaign

Verification of Imperative Programs

We are now ready to consider the verification of programs in an imperative language. We will do so using a simple imperative language called IMPL computing with both numbers and lists.

Verification of Imperative Programs

We are now ready to consider the verification of programs in an imperative language. We will do so using a simple imperative language called IMPL computing with both numbers and lists.

Of course, for the formal verification of some properties Q about a program P in an imperative language $\mathcal{L}$ to be meaningful at all, our first and most crucial task is to make sure that the programming language $\mathcal{L}$ has a clear and precise mathematical semantics, since only then can we settle mathematically whether a program P satisfies some properties Q.

Verification of Imperative Programs (II)

The issue of giving a mathematical semantics to an imperative programming language $\mathcal{L}$ is actually nontrivial.

Verification of Imperative Programs (II)

The issue of giving a mathematical semantics to an imperative programming language $\mathcal{L}$ is actually nontrivial. It is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

Verification of Imperative Programs (II)

The issue of giving a mathematical semantics to an imperative programming language $\mathcal{L}$ is actually nontrivial. It is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

For example, for a Maude functional module, its mathematical semantics is the initial algebra of its equational theory.

Verification of Imperative Programs (II)

The issue of giving a mathematical semantics to an imperative programming language $\mathcal{L}$ is actually nontrivial. It is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

For example, for a Maude functional module, its mathematical semantics is the initial algebra of its equational theory. And its operational semantics is given by equational simplification with its equations, which are assumed confluent and terminating.

Verification of Imperative Programs (II)

The issue of giving a mathematical semantics to an imperative programming language $\mathcal{L}$ is actually nontrivial. It is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

For example, for a Maude functional module, its mathematical semantics is the initial algebra of its equational theory. And its operational semantics is given by equational simplification with its equations, which are assumed confluent and terminating.

Some imperative languages have never been given a formal semantics.

Verification of Imperative Programs (II)

The issue of giving a mathematical semantics to an imperative programming language $\mathcal{L}$ is actually nontrivial. It is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

For example, for a Maude functional module, its mathematical semantics is the initial algebra of its equational theory. And its operational semantics is given by equational simplification with its equations, which are assumed confluent and terminating.

Some imperative languages have never been given a formal semantics.Their only precise documentation may be the different compilers, perhaps inconsistent with each other.

Verification of Imperative Programs (III)

Giving mathematical semantics to an imperative language $\mathcal{L}$ amounts to defining a mathematical model of the language.

Verification of Imperative Programs (III)

Giving mathematical semantics to an imperative language $\mathcal{L}$ amounts to defining a mathematical model of the language. This is done using some mathematical formalism: either set theory, which is a de-facto universal formalism for mathematics, or some other formalism.

Verification of Imperative Programs (III)

Giving mathematical semantics to an imperative language $\mathcal{L}$ amounts to defining a mathematical model of the language. This is done using some mathematical formalism: either set theory, which is a de-facto universal formalism for mathematics, or some other formalism.

In practice, however, the choice of formalism is crucial.

Verification of Imperative Programs (III)

Giving mathematical semantics to an imperative language $\mathcal{L}$ amounts to defining a mathematical model of the language. This is done using some mathematical formalism: either set theory, which is a de-facto universal formalism for mathematics, or some other formalism.

In practice, however, the choice of formalism is crucial. For example, a large language like C had no complete formal semantics until 2012.

Verification of Imperative Programs (III)

Giving mathematical semantics to an imperative language $\mathcal{L}$ amounts to defining a mathematical model of the language. This is done using some mathematical formalism: either set theory, which is a de-facto universal formalism for mathematics, or some other formalism.

In practice, however, the choice of formalism is crucial. For example, a large language like C had no complete formal semantics until 2012. It was given by Chucky Ellison and Grigore Roşu from UIUC at the POPL 2012 Conference as a rewrite theory $\mathcal{R}_C$, which was desugared into a Maude module by their K Tool for execution purposes.

Verification of Imperative Programs (III)

Giving mathematical semantics to an imperative language $\mathcal{L}$ amounts to defining a mathematical model of the language. This is done using some mathematical formalism: either set theory, which is a de-facto universal formalism for mathematics, or some other formalism.

In practice, however, the choice of formalism is crucial. For example, a large language like C had no complete formal semantics until 2012. It was given by Chucky Ellison and Grigore Roşu from UIUC at the POPL 2012 Conference as a rewrite theory $\mathcal{R}_C$, which was desugared into a Maude module by their K Tool for execution purposes. Executability of $\mathcal{R}_C$ was crucial: otherwise, a paper semantics of C could be totally wrong.

Verification of Imperative Programs (III)

Giving mathematical semantics to an imperative language $\mathcal{L}$ amounts to defining a mathematical model of the language. This is done using some mathematical formalism: either set theory, which is a de-facto universal formalism for mathematics, or some other formalism.

In practice, however, the choice of formalism is crucial. For example, a large language like C had no complete formal semantics until 2012. It was given by Chucky Ellison and Grigore Roşu from UIUC at the POPL 2012 Conference as a rewrite theory $\mathcal{R}_C$, which was desugared into a Maude module by their K Tool for execution purposes. Executability of $\mathcal{R}_C$ was crucial: otherwise, a paper semantics of C could be totally wrong. This was an important step in the Rewriting Logic Semantics Project, started by Meseguer and Roşu in 2004.

Rewriting Logic Semantics of Imperative Languages

The rewriting logic semantics of an imperative language $\mathcal{L}$ is given by a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup B, R_{\mathcal{L}})$ such that:

Rewriting Logic Semantics of Imperative Languages

The rewriting logic semantics of an imperative language $\mathcal{L}$ is given by a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup B, R_{\mathcal{L}})$ such that:

- The syntax of $\mathcal{L}$ is specified as a subsignature of $\Sigma_{\mathcal{L}}$.

Rewriting Logic Semantics of Imperative Languages

The rewriting logic semantics of an imperative language $\mathcal{L}$ is given by a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup B, R_{\mathcal{L}})$ such that:

- The syntax of $\mathcal{L}$ is specified as a subsignature of $\Sigma_{\mathcal{L}}$.
- The mathematical semantics of $\mathcal{L}$ is given by the initial reachability model $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$.

Rewriting Logic Semantics of Imperative Languages

The rewriting logic semantics of an imperative language $\mathcal{L}$ is given by a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup B, R_{\mathcal{L}})$ such that:

- The syntax of $\mathcal{L}$ is specified as a subsignature of $\Sigma_{\mathcal{L}}$.
- The mathematical semantics of $\mathcal{L}$ is given by the initial reachability model $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$.
- The operational semantics of $\mathcal{L}$ is given by the rewrite rules $R_{\mathcal{L}}$, called the semantic rules of $\mathcal{L}$, which are coherent with the convergent equations $E_{\mathcal{L}}$ modulo B.

Rewriting Logic Semantics of Imperative Languages

The rewriting logic semantics of an imperative language $\mathcal{L}$ is given by a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup B, R_{\mathcal{L}})$ such that:

- The syntax of $\mathcal{L}$ is specified as a subsignature of $\Sigma_{\mathcal{L}}$.
- The mathematical semantics of $\mathcal{L}$ is given by the initial reachability model $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$.
- The operational semantics of $\mathcal{L}$ is given by the rewrite rules $R_{\mathcal{L}}$, called the semantic rules of $\mathcal{L}$, which are coherent with the convergent equations $E_{\mathcal{L}}$ modulo B.

We can illustrate this general approach, applicable to any imperative language $\mathcal{L}$, by considering the IMPL language.

Definitional Styles

The semantic definition of a language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be given in very different definitional styles,[1] such as:

[1] See T. Serbanuta, G. Roşu and J. Meseguer, *A Rewriting Logic Approach to Operational Semantics*, *Inf. & Comput.*, 207, 305–340, 2009.

Definitional Styles

The semantic definition of a language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be given in very different definitional styles,[1] such as:

- small step or big step semantics,

[1] See T. Serbanuta, G. Roşu and J. Meseguer, *A Rewriting Logic Approach to Operational Semantics*, *Inf. & Comput.*, 207, 305–340, 2009.

Definitional Styles

The semantic definition of a language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be given in very different definitional styles,[1] such as:

- small step or big step semantics,
- reduction semantics,

[1] See T. Serbanuta, G. Roşu and J. Meseguer, *A Rewriting Logic Approach to Operational Semantics*, *Inf. & Comput.*, 207, 305–340, 2009.

Definitional Styles

The semantic definition of a language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be given in very different definitional styles,[1] such as:

- small step or big step semantics,
- reduction semantics,
- continuation semantics, and

[1] See T. Serbanuta, G. Roşu and J. Meseguer, *A Rewriting Logic Approach to Operational Semantics*, *Inf. & Comput.*, 207, 305–340, 2009.

Definitional Styles

The semantic definition of a language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be given in very different definitional styles,[1] such as:

- small step or big step semantics,
- reduction semantics,
- continuation semantics, and
- MSOS or CHAM semantics.

[1] See T. Serbanuta, G. Roşu and J. Meseguer, *A Rewriting Logic Approach to Operational Semantics*, *Inf. & Comput.*, 207, 305–340, 2009.

Definitional Styles

The semantic definition of a language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be given in very different definitional styles,[1] such as:

- small step or big step semantics,
- reduction semantics,
- continuation semantics, and
- MSOS or CHAM semantics.

The semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$ will use a continuation style, because it is simple, flexible, efficient, and highly scalable.

[1] See T. Serbanuta, G. Roşu and J. Meseguer, *A Rewriting Logic Approach to Operational Semantics*, *Inf. & Comput.*, 207, 305–340, 2009.

Definitional Styles

The semantic definition of a language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be given in very different definitional styles,[1] such as:

- small step or big step semantics,
- reduction semantics,
- continuation semantics, and
- MSOS or CHAM semantics.

The semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$ will use a continuation style, because it is simple, flexible, efficient, and highly scalable. For example, Ellison and Roşu's C semantics is continuation style.

[1] See T. Serbanuta, G. Roşu and J. Meseguer, *A Rewriting Logic Approach to Operational Semantics*, *Inf. & Comput.*, 207, 305–340, 2009.

IMPL Syntax

Identifiers are defined as follows:

$$Id ::= a \mid b \mid c \mid i \mid j \mid k \mid x \mid y \mid z \mid Id,$$

Identifiers can be built from basic ones using the comma operator.

Data. We have $Bool$, Nat, and $List$ data built from constructors.

$Bool$ data is defined as expected:

$$Bool ::= true \mid false$$

Nat uses constructors 0, 1 and $+$, with $+$ AC and 0 as its identity:

$$Nat ::= 0 \mid 1 \mid Nat \ + \ Nat$$

$List$ contains Nat as a subtype and has two constructors, nil and an associative list concatenation constructor $\_\$\_$.

$$List ::= nil \mid Nat \mid List \ \mathbf{\$} \ List$$

IMPL Syntax (II)

Expressions IMPL has arithmetic, list, and Boolean expressions.

Arithmetic expressions, or $AExp$, have syntax:

$$AExp ::= Id \mid Nat \mid AExp +: AExp \mid AExp -: AExp \mid AExp *: AExp$$

List expressions, or $LExp$ have syntax:

$$LExp ::= Id \mid List \mid LExp \,\$: LExp \mid \mathit{first}(LExp) \mid \mathit{rest}(LExp)$$

Boolean expressions, or $BExp$, have syntax:

$$BExp ::= Bool \mid !\, BExp \mid AExp <: AExp \mid BExp \; and \; BExp \mid \mathit{empty}(LExp)$$

IMPL Syntax (III)

Statements have the following syntax:

$$
\begin{aligned}
Stmt ::= & \ \{\ \} \\
& \mid Stmt\ Stmt \\
& \mid \{\ Stmt\ \} \\
& \mid Id\ =\ AExp\ ; \\
& \mid Id\ =_l\ LExp\ ; \\
& \mid \textbf{while}\ (BExp)\ Stmt \\
& \mid \textbf{if}\ (BExp)\ Stmt\ \textbf{else}\ Stmt
\end{aligned}
$$

The empty statement is denoted by two curly brackets, denoting "skip."

IMPL Syntax (III)

Statements have the following syntax:

$$
\begin{aligned}
Stmt ::= & \ \{\ \} \\
& \mid Stmt\ Stmt \\
& \mid \{\ Stmt\ \} \\
& \mid Id\ =\ AExp\ ; \\
& \mid Id\ =_l\ LExp\ ; \\
& \mid \textbf{while}\ (BExp)\ Stmt \\
& \mid \textbf{if}\ (BExp)\ Stmt\ \textbf{else}\ Stmt
\end{aligned}
$$

The empty statement is denoted by two curly brackets, denoting "skip." Statement concatenation is denoted $Stmt\ Stmt$.

IMPL Syntax (III)

Statements have the following syntax:

$$
\begin{aligned}
Stmt ::= & \ \{\ \} \\
& \mid Stmt\ Stmt \\
& \mid \{\ Stmt\ \} \\
& \mid Id\ =\ AExp\ ; \\
& \mid Id\ =_l\ LExp\ ; \\
& \mid \textbf{while}\ (BExp)\ Stmt \\
& \mid \textbf{if}\ (BExp)\ Stmt\ \textbf{else}\ Stmt
\end{aligned}
$$

The empty statement is denoted by two curly brackets, denoting "skip." Statement concatenation is denoted $Stmt\ Stmt$.
Arithmetic assignments use the standard equals character.

IMPL Syntax (III)

Statements have the following syntax:

$$
\begin{aligned}
Stmt ::= & \ \{\ \} \\
& \mid Stmt\ Stmt \\
& \mid \{\ Stmt\ \} \\
& \mid Id\ =\ AExp\ ; \\
& \mid Id\ =_l\ LExp\ ; \\
& \mid \textbf{while}\ (BExp)\ Stmt \\
& \mid \textbf{if}\ (BExp)\ Stmt\ \textbf{else}\ Stmt
\end{aligned}
$$

The empty statement is denoted by two curly brackets, denoting "skip." Statement concatenation is denoted $Stmt\ Stmt$. Arithmetic assignments use the standard equals character. List assignments require the letter *l* after the '='.

IMPL Syntax (III)

Statements have the following syntax:

$$
\begin{aligned}
Stmt ::= & \ \{\ \} \\
& \mid Stmt\ Stmt \\
& \mid \{\ Stmt\ \} \\
& \mid Id\ =\ AExp\ ; \\
& \mid Id\ =_l\ LExp\ ; \\
& \mid \textbf{while}\ (BExp)\ Stmt \\
& \mid \textbf{if}\ (BExp)\ Stmt\ \textbf{else}\ Stmt
\end{aligned}
$$

The empty statement is denoted by two curly brackets, denoting "skip." Statement concatenation is denoted $Stmt\ Stmt$. Arithmetic assignments use the standard equals character. List assignments require the letter *l* after the '='. While loops and conditionals have standard syntax.

IMPL Continuation Semantics

In the IMPL continuation semantics, computation states are pairs:

$$< Continuation \mid Store >$$

where $Continuation$ represents the rest of the program which remains to be executed, and $Store$ is the current store, mapping program variables to their current values.

IMPL Continuation Semantics

In the IMPL continuation semantics, computation states are pairs:

$$< Continuation \mid Store >$$

where $Continuation$ represents the rest of the program which remains to be executed, and $Store$ is the current store, mapping program variables to their current values. In an initial state the first component has the form $P \rightsquigarrow done$, with P the program to be executed and $done$ a stopping continuation.

IMPL Continuation Semantics

In the IMPL continuation semantics, computation states are pairs:

$$< Continuation \mid Store >$$

where $Continuation$ represents the rest of the program which remains to be executed, and $Store$ is the current store, mapping program variables to their current values. In an initial state the first component has the form $P \rightsquigarrow done$, with P the program to be executed and $done$ a stopping continuation. The second second component is the initial store.

IMPL Continuation Semantics

In the IMPL continuation semantics, computation states are pairs:

$$< Continuation \mid Store >$$

where $Continuation$ represents the rest of the program which remains to be executed, and $Store$ is the current store, mapping program variables to their current values. In an initial state the first component has the form $P \rightsquigarrow done$, with P the program to be executed and $done$ a stopping continuation. The second second component is the initial store.

Before program execution begins, $P \rightsquigarrow done$ is transformed into a sequence of elementary tasks of form:

$$T_1 \rightsquigarrow T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1} \rightsquigarrow done$$

which are then executed from left to right by the semantic rules R_{IMPL}.

IMPL Continuation Semantics

In the IMPL continuation semantics, computation states are pairs:

$$< Continuation \mid Store >$$

where $Continuation$ represents the rest of the program which remains to be executed, and $Store$ is the current store, mapping program variables to their current values. In an initial state the first component has the form $P \rightsquigarrow done$, with P the program to be executed and $done$ a stopping continuation. The second second component is the initial store.

Before program execution begins, $P \rightsquigarrow done$ is transformed into a sequence of elementary tasks of form:

$$T_1 \rightsquigarrow T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1} \rightsquigarrow done$$

which are then executed from left to right by the semantic rules R_{IMPL}. The tranformation of $P \rightsquigarrow done$ into a sequence of tasks is achieved by the following equations in E_{IMPL}:

IMPL Continuation Semantics (II)

$$(X = AE;) \rightsquigarrow K = AE \rightsquigarrow \ =(X) \ \rightsquigarrow K$$
$$(X =_l LE;) \rightsquigarrow K = LE \rightsquigarrow \ =(X) \ \rightsquigarrow K$$

$$S\ S' \rightsquigarrow K \ = \ S \rightsquigarrow S' \rightsquigarrow K$$
$$\{S\} \rightsquigarrow K \ = \ S \rightsquigarrow K$$
$$\{\} \rightsquigarrow K \ = \ K$$

$$\textbf{if}\ (B)\ S\ \textbf{else}\ S' \rightsquigarrow K \ = \ B \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K$$
$$\textbf{while}\ (BE)\ \{S\} \rightsquigarrow K \ = \ BE \rightsquigarrow \textbf{if}(\{S\ \textbf{while}\ (BE)\ \{S\}\}\ ,\ \{\}) \rightsquigarrow K$$

IMPL Continuation Semantics (III)

$$AE_1 + : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow \; + : \rightsquigarrow K$$
$$AE_1 * : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow \; * : \rightsquigarrow K$$
$$AE_1 - : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow \; - : \rightsquigarrow K$$
$$AE_1 < : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow < : \rightsquigarrow K$$

$$!BE \rightsquigarrow K \;=\; BE \rightsquigarrow ! \rightsquigarrow K$$
$$BE_1 \; and \; BE_2 \rightsquigarrow K \;=\; BE_1 \rightsquigarrow and(BE_2) \rightsquigarrow K$$

$$LE_1 \; \$: LE_2 \rightsquigarrow K \;=\; (LE_1, LE_2) \rightsquigarrow \$: \rightsquigarrow K$$
$$\mathit{first}(LE) \rightsquigarrow K \;=\; LE \rightsquigarrow \mathit{first} \rightsquigarrow K$$
$$\mathit{rest}(LE) \rightsquigarrow K \;=\; LE \rightsquigarrow \mathit{rest} \rightsquigarrow K$$
$$\mathit{empty}(LE) \rightsquigarrow K \;=\; LE \rightsquigarrow \mathit{empty} \rightsquigarrow K$$

IMPL Continuation Semantics (III)

$$AE_1 + : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow \; + : \rightsquigarrow K$$
$$AE_1 * : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow \; * : \rightsquigarrow K$$
$$AE_1 - : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow \; - : \rightsquigarrow K$$
$$AE_1 < : AE_2 \rightsquigarrow K \;=\; (AE_1, AE_2) \rightsquigarrow < : \rightsquigarrow K$$

$$!BE \rightsquigarrow K \;=\; BE \rightsquigarrow ! \rightsquigarrow K$$
$$BE_1 \; and \; BE_2 \rightsquigarrow K \;=\; BE_1 \rightsquigarrow and(BE_2) \rightsquigarrow K$$

$$LE_1 \; \$: LE_2 \rightsquigarrow K \;=\; (LE_1, LE_2) \rightsquigarrow \$: \rightsquigarrow K$$
$$first(LE) \rightsquigarrow K \;=\; LE \rightsquigarrow first \rightsquigarrow K$$
$$rest(LE) \rightsquigarrow K \;=\; LE \rightsquigarrow rest \rightsquigarrow K$$
$$empty(LE) \rightsquigarrow K \;=\; LE \rightsquigarrow empty \rightsquigarrow K$$

$T_1 \rightsquigarrow T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1} \rightsquigarrow done$ is the canonical form of $P \rightsquigarrow done$ by the above equations.

Stores

The store contains the current state of the program's variables, together with type information.

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$.

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$.

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

$$x \mapsto 1 * y \mapsto (1 + 1) \,\$\, 1 \,\$\, (1 + 1 + 1)$$

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

$$x \mapsto 1 * y \mapsto (1+1)\,\$\,1\,\$\,(1+1+1)$$

A $TStore$ (for *Type Store*), records type information.

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

$$x \mapsto 1 * y \mapsto (1 + 1) \,\$\, 1 \,\$\, (1 + 1 + 1)$$

A $TStore$ (for *Type Store*), records type information. A $TStore$ is like a $VStore$, but identifiers are now mapped to types: either to $TNat$ or $TList$.

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

$$x \mapsto 1 * y \mapsto (1+1) \,\$\, 1 \,\$\, (1+1+1)$$

A $TStore$ (for *Type Store*), records type information. A $TStore$ is like a $VStore$, but identifiers are now mapped to types: either to $TNat$ or $TList$. A full store $Store$ is a pair $TStore \,\&\, VStore$ with $TStore$ the type store of $VStore$.

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

$$x \mapsto 1 * y \mapsto (1 + 1) \,\$\, 1 \,\$\, (1 + 1 + 1)$$

A $TStore$ (for *Type Store*), records type information. A $TStore$ is like a $VStore$, but identifiers are now mapped to types: either to $TNat$ or $TList$. A full store $Store$ is a pair $TStore \,\&\, VStore$ with $TStore$ the type store of $VStore$. For our example the full store is:

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

$$x \mapsto 1 * y \mapsto (1+1) \,\$\, 1 \,\$\, (1+1+1)$$

A $TStore$ (for *Type Store*), records type information. A $TStore$ is like a $VStore$, but identifiers are now mapped to types: either to $TNat$ or $TList$. A full store $Store$ is a pair $TStore \,\&\, VStore$ with $TStore$ the type store of $VStore$. For our example the full store is:

$$x \mapsto TNat * y \mapsto TList \,\&\, x \mapsto 1 * y \mapsto (1{+}1) \,\$\, 1 \,\$\, (1{+}1{+}1)$$

Stores

The store contains the current state of the program's variables, together with type information. A mapping of an identifier x to either a Nat or $List$ data value v is a pair $x \mapsto v$. A $VStore$ (for *Value Store*), is a finite function, i.e., a set of such pairs built up with the AC union operator $\_ * \_$. The following is a valid store:

$$x \mapsto 1 * y \mapsto (1+1) \,\$\, 1 \,\$\, (1+1+1)$$

A $TStore$ (for *Type Store*), records type information. A $TStore$ is like a $VStore$, but identifiers are now mapped to types: either to $TNat$ or $TList$. A full store $Store$ is a pair $TStore \,\&\, VStore$ with $TStore$ the type store of $VStore$. For our example the full store is:

$$x \mapsto TNat * y \mapsto TList \,\&\, x \mapsto 1 * y \mapsto (1+1) \,\$\, 1 \,\$\, (1+1+1)$$

$mtVE$, $mtTE$, and mt to denote the empty state, empty type store, and empty full store.

IMPL Semantic Rules

The semantic rules R_{IMPL} in $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$ perform computation steps that execute the top task of the current continuation sequence in the current store. They are as follows (some similar rules omitted):

IMPL Semantic Rules

The semantic rules R_{IMPL} in $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$ perform computation steps that execute the top task of the current continuation sequence in the current store. They are as follows (some similar rules omitted):

Variable Update and Variable Lookup Semantic Rules. The rules for update and lookup of variables of type $TNat$ are given below.

IMPL Semantic Rules

The semantic rules R_{IMPL} in $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$ perform computation steps that execute the top task of the current continuation sequence in the current store. They are as follows (some similar rules omitted):

Variable Update and Variable Lookup Semantic Rules. The rules for update and lookup of variables of type $TNat$ are given below. Similar rules handle variables of type $TList$.

IMPL Semantic Rules

The semantic rules R_{IMPL} in $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$ perform computation steps that execute the top task of the current continuation sequence in the current store. They are as follows (some similar rules omitted):

Variable Update and Variable Lookup Semantic Rules. The rules for update and lookup of variables of type $TNat$ are given below. Similar rules handle variables of type $TList$.

$$
\begin{array}{ll}
< N \rightsquigarrow =(X) \rightsquigarrow K & \mid (TSt * (X \mapsto TNat)) \,\&\, (VSt * (X \mapsto N')) > \\
\rightarrow < K & \mid (TSt * (X \mapsto TNat)) \,\&\, (VSt * (X \mapsto N)) > \\
< X \rightsquigarrow K & \mid (TSt * (X \mapsto TNat)) \,\&\, (VSt * (X \mapsto N)) > \\
\rightarrow < N \rightsquigarrow K & \mid (TSt * (X \mapsto TNat)) \,\&\, (VSt * (X \mapsto N)) >
\end{array}
$$

IMPL Semantic Rules (II)

Arithmetic, Boolean, and List Data Type Rules. The semantic rules below perform elementary operations for natural numbers and for Booleans (there are similar rules for lists). The operation symbols on the righthand sides, like $+$, $*$ and so on, are performed in the associated data types for naturals, Booleans and lists.

$$< (I_1, I_2) \rightsquigarrow +: \rightsquigarrow K \mid St > \rightarrow < I_1 + I_2 \rightsquigarrow K \mid St >$$
$$< (I_1, I_2) \rightsquigarrow *: \rightsquigarrow K \mid St > \rightarrow < I_1 * I_2 \rightsquigarrow K \mid St >$$
$$< true \rightsquigarrow ! \rightsquigarrow K \mid St > \rightarrow < false \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow ! \rightsquigarrow K \mid St > \rightarrow < true \rightsquigarrow K \mid St >$$
$$< true \rightsquigarrow \textbf{and}(BE) \rightsquigarrow K \mid St > \rightarrow < BE \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{and}(BE) \rightsquigarrow K \mid St > \rightarrow < false \rightsquigarrow K \mid St >$$

Interlude: Tuple Continuation Equations

Except for Boolean conjunction, all other binary operations in expressions are handled the same way.

Interlude: Tuple Continuation Equations

Except for Boolean conjunction, all other binary operations in expressions are handled the same way. A continuation $AE_1\, op\colon AE_2 \rightsquigarrow K$ is transformed by the E_{IMPL} equations into a continuation $(AE_1, AE_2) \rightsquigarrow op{:}\!\rightsquigarrow K$.

Interlude: Tuple Continuation Equations

Except for Boolean conjunction, all other binary operations in expressions are handled the same way. A continuation $AE_1\, op\colon AE_2 \rightsquigarrow K$ is transformed by the E_{IMPL} equations into a continuation $(AE_1, AE_2) \rightsquigarrow op{:}\!\rightsquigarrow K$. But how is the 2-tuple (AE_1, AE_2) evaluated to its values (I_1, I_2)?

Interlude: Tuple Continuation Equations

Except for Boolean conjunction, all other binary operations in expressions are handled the same way. A continuation $AE_1\, op\colon AE_2 \rightsquigarrow K$ is transformed by the E_{IMPL} equations into a continuation $(AE_1, AE_2) \rightsquigarrow op{:}\!\rightsquigarrow K$. But how is the 2-tuple (AE_1, AE_2) evaluated to its values (I_1, I_2)? And how is this done in a left-to-right evaluation order?

Interlude: Tuple Continuation Equations

Except for Boolean conjunction, all other binary operations in expressions are handled the same way. A continuation $AE_1\, op\colon AE_2 \rightsquigarrow K$ is transformed by the E_{IMPL} equations into a continuation $(AE_1, AE_2) \rightsquigarrow op\colon\!\rightsquigarrow K$. But how is the 2-tuple (AE_1, AE_2) evaluated to its values (I_1, I_2)? And how is this done in a left-to-right evaluation order? This is where tuple continuation equations come in.

Interlude: Tuple Continuation Equations

Except for Boolean conjunction, all other binary operations in expressions are handled the same way. A continuation $AE_1\, op\colon AE_2 \rightsquigarrow K$ is transformed by the E_{IMPL} equations into a continuation $(AE_1, AE_2) \rightsquigarrow op{:}\!\rightsquigarrow K$. But how is the 2-tuple (AE_1, AE_2) evaluated to its values (I_1, I_2)? And how is this done in a left-to-right evaluation order? This is where tuple continuation equations come in. For arithmetic expressions they are (similar equations handle lists):

Interlude: Tuple Continuation Equations

Except for Boolean conjunction, all other binary operations in expressions are handled the same way. A continuation $AE_1\, op\colon AE_2 \rightsquigarrow K$ is transformed by the E_{IMPL} equations into a continuation $(AE_1, AE_2) \rightsquigarrow op{:}\!\rightsquigarrow K$. But how is the 2-tuple (AE_1, AE_2) evaluated to its values (I_1, I_2)? And how is this done in a left-to-right evaluation order? This is where tuple continuation equations come in. For arithmetic expressions they are (similar equations handle lists):

$$
\begin{aligned}
(AE_1, AE_2) \rightsquigarrow K &= AE_1 \rightsquigarrow (\#, AE_2) \rightsquigarrow K \\
I_1 \rightsquigarrow (\#, AE_2) \rightsquigarrow K &= AE_2 \rightsquigarrow (I_1, \#) \rightsquigarrow K \\
I_2 \rightsquigarrow (I_1, \#) \rightsquigarrow K &= (I_1, I_2) \rightsquigarrow K
\end{aligned}
$$

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition.

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

$$< true \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >$$

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

$$< true \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >$$

In this continuation style, the semantic rules R_{IMPL} are extremely simple:

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

$$< true \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >$$

In this continuation style, the semantic rules R_{IMPL} are extremely simple: they just perform elementary computation steps.

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

$$< true \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >$$

In this continuation style, the semantic rules R_{IMPL} are extremely simple: they just perform elementary computation steps. The secret of this simplicity is the transformation by the equations E_{IMPL} of the initial $P \rightsquigarrow done$ into a sequence of simple tasks.

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

$$< true \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >$$

In this continuation style, the semantic rules R_{IMPL} are extremely simple: they just perform elementary computation steps. The secret of this simplicity is the transformation by the equations E_{IMPL} of the initial $P \rightsquigarrow done$ into a sequence of simple tasks.

The only slightly more subtle equation is the recursive one for while loops, $while\ (BE)\ \{S\} \rightsquigarrow K\ =\ BE \rightsquigarrow$ $if\,(\{S\ while\ (BE)\ \{S\}\}\ ,\ \{\}) \rightsquigarrow K$.

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

$$< true \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >$$

In this continuation style, the semantic rules R_{IMPL} are extremely simple: they just perform elementary computation steps. The secret of this simplicity is the transformation by the equations E_{IMPL} of the initial $P \rightsquigarrow done$ into a sequence of simple tasks.

The only slightly more subtle equation is the recursive one for while loops, $while\ (BE)\ \{S\} \rightsquigarrow K\ =\ BE \rightsquigarrow$ $if\,(\{S\ while\ (BE)\ \{S\}\}\ ,\ \{\}) \rightsquigarrow K$. After BE gets evaluated to either *true* or *false*, this triggers the application of one of the two branching rules above.

IMPL Semantic Rules (and III)

Branching Semantic Rules. The only remaining rules are the two ones for branching on a condition. They are applied after a Boolean condition B gets evaluated to *true* or *false*.

$$< true \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St >$$
$$< false \rightsquigarrow \textbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >$$

In this continuation style, the semantic rules R_{IMPL} are extremely simple: they just perform elementary computation steps. The secret of this simplicity is the transformation by the equations E_{IMPL} of the initial $P \rightsquigarrow done$ into a sequence of simple tasks.

The only slightly more subtle equation is the recursive one for while loops, $while\ (BE)\ \{S\} \rightsquigarrow K\ =\ BE \rightsquigarrow$ $if\,(\{S\ while\ (BE)\ \{S\}\}\ ,\ \{\}) \rightsquigarrow K$. After BE gets evaluated to either *true* or *false*, this triggers the application of one of the two branching rules above. This is what makes IMPL Turing complete.

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$.

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$. The equations E_{IMPL} abstract away all the syntactic manipulations of a program's abstract syntax tree to transform it into a list of tasks.

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$. The equations E_{IMPL} abstract away all the syntactic manipulations of a program's abstract syntax tree to transform it into a list of tasks. Since this is done modulo E_{IMPL} such manipulations become invisible. How can make them visible?

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$. The equations E_{IMPL} abstract away all the syntactic manipulations of a program's abstract syntax tree to transform it into a list of tasks. Since this is done modulo E_{IMPL} such manipulations become invisible. How can make them visible? By transforming the equations E_{IMPL} into rules.

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$. The equations E_{IMPL} abstract away all the syntactic manipulations of a program's abstract syntax tree to transform it into a list of tasks. Since this is done modulo E_{IMPL} such manipulations become invisible. How can make them visible? By transforming the equations E_{IMPL} into rules. All such equations have the form $k = k'$, with k, k' continuation expressions.

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$. The equations E_{IMPL} abstract away all the syntactic manipulations of a program's abstract syntax tree to transform it into a list of tasks. Since this is done modulo E_{IMPL} such manipulations become invisible. How can make them visible? By transforming the equations E_{IMPL} into rules. All such equations have the form $k = k'$, with k, k' continuation expressions. Instead of transforming them into rules $k \rightarrow k'$ it is better to make then semantic rules $< k \mid St > \rightarrow < k' \mid St >$. Let $\vec{E}^{\bullet}_{IMPL}$ denote such rules.

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$. The equations E_{IMPL} abstract away all the syntactic manipulations of a program's abstract syntax tree to transform it into a list of tasks. Since this is done modulo E_{IMPL} such manipulations become invisible. How can make them visible? By transforming the equations E_{IMPL} into rules. All such equations have the form $k = k'$, with k, k' continuation expressions. Instead of transforming them into rules $k \rightarrow k'$ it is better to make then semantic rules $< k \mid St > \rightarrow < k' \mid St >$. Let $\vec{E}^{\bullet}_{IMPL}$ denote such rules. Then

$$\mathcal{R}^{FG}_{IMPL} = (\Sigma_{IMPL}, B, \vec{E}^{\bullet}_{IMPL} \cup R_{IMPL})$$

Abstract vs. Fine-Grained Continuation Semantics

We have just specified the continuation semantics of IMPL as a rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$. The equations E_{IMPL} abstract away all the syntactic manipulations of a program's abstract syntax tree to transform it into a list of tasks. Since this is done modulo E_{IMPL} such manipulations become invisible. How can make them visible? By transforming the equations E_{IMPL} into rules. All such equations have the form $k = k'$, with k, k' continuation expressions. Instead of transforming them into rules $k \rightarrow k'$ it is better to make then semantic rules $< k \mid St > \rightarrow < k' \mid St >$. Let $\vec{E}^{\bullet}_{IMPL}$ denote such rules. Then

$$\mathcal{R}^{FG}_{IMPL} = (\Sigma_{IMPL}, B, \vec{E}^{\bullet}_{IMPL} \cup R_{IMPL})$$

is the fine-grained semantics of IMPL.

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P?

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P? By specifying such properties as reachability formulas and proving them in the Reachability Logic Prover (RLP).

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P? By specifying such properties as reachability formulas and proving them in the Reachability Logic Prover (RLP). How so?

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P? By specifying such properties as reachability formulas and proving them in the Reachability Logic Prover (RLP). How so? RLP is theory-generic. It can prove reachability properties about any rewrite theory $\mathcal{R}$ under fairly mild assumptions.

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P? By specifying such properties as reachability formulas and proving them in the Reachability Logic Prover (RLP). How so? RLP is theory-generic. It can prove reachability properties about any rewrite theory $\mathcal{R}$ under fairly mild assumptions. We just:

1. instantiate RLP with the rewrite theory $\mathcal{R}^{FG}_{IMPL}$ defining the semantics of IMPL, and

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P? By specifying such properties as reachability formulas and proving them in the Reachability Logic Prover (RLP). How so? RLP is theory-generic. It can prove reachability properties about any rewrite theory $\mathcal{R}$ under fairly mild assumptions. We just:

1. instantiate RLP with the rewrite theory $\mathcal{R}^{FG}_{IMPL}$ defining the semantics of IMPL, and
2. specify the properties of a program P in IMPL as reachability formulas in $\mathcal{R}_{IMPL}$.

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P? By specifying such properties as reachability formulas and proving them in the Reachability Logic Prover (RLP). How so? RLP is theory-generic. It can prove reachability properties about any rewrite theory $\mathcal{R}$ under fairly mild assumptions. We just:

1. instantiate RLP with the rewrite theory $\mathcal{R}^{FG}_{IMPL}$ defining the semantics of IMPL, and
2. specify the properties of a program P in IMPL as reachability formulas in $\mathcal{R}_{IMPL}$.

How do such formulas look like for $\mathcal{R}_{IMPL}$?

Proving Properties of IMPL Programs

Now that we have a formal semantics of IMPL as a rewrite theory $\mathcal{R}$, how can we prove properties of an IMPL program P? By specifying such properties as reachability formulas and proving them in the Reachability Logic Prover (RLP). How so? RLP is theory-generic. It can prove reachability properties about any rewrite theory $\mathcal{R}$ under fairly mild assumptions. We just:

1. instantiate RLP with the rewrite theory $\mathcal{R}^{FG}_{IMPL}$ defining the semantics of IMPL, and
2. specify the properties of a program P in IMPL as reachability formulas in $\mathcal{R}_{IMPL}$.

How do such formulas look like for $\mathcal{R}_{IMPL}$? Let us see.

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$,

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

$$< P \rightsquigarrow done \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < done \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

$$< P \rightsquigarrow done \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < done \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

where $\vec{x} = x_1, \ldots, x_n$ are the program variables in P,

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

$$< P \rightsquigarrow done \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < done \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

where $\vec{x} = x_1, \ldots, x_n$ are the program variables in P, and $\vec{x} \mapsto \vec{X}$ (similar for $\vec{x} \mapsto \vec{X'}$) abbreviates the $VStore$ fragment: $x_1 \mapsto X_1 * \ldots * x_n \mapsto X_n$.

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

$$< P \rightsquigarrow done \mid TS\ \&\ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < done \mid TS\ \&\ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

where $\vec{x} = x_1, \ldots, x_n$ are the program variables in P, and $\vec{x} \mapsto \vec{X}$ (similar for $\vec{x} \mapsto \vec{X'}$) abbreviates the $VStore$ fragment: $x_1 \mapsto X_1 * \ldots * x_n \mapsto X_n$. It will always be useful to generalize such a Hoare formula to the general reachability formula:

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

$$< P \rightsquigarrow done \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < done \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

where $\vec{x} = x_1, \ldots, x_n$ are the program variables in P, and $\vec{x} \mapsto \vec{X}$ (similar for $\vec{x} \mapsto \vec{X'}$) abbreviates the $VStore$ fragment: $x_1 \mapsto X_1 * \ldots * x_n \mapsto X_n$. It will always be useful to generalize such a Hoare formula to the general reachability formula:

$$< P \rightsquigarrow K \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < K \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

$$< P \rightsquigarrow done \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < done \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

where $\vec{x} = x_1, \ldots, x_n$ are the program variables in P, and $\vec{x} \mapsto \vec{X}$ (similar for $\vec{x} \mapsto \vec{X'}$) abbreviates the $VStore$ fragment: $x_1 \mapsto X_1 * \ldots * x_n \mapsto X_n$. It will always be useful to generalize such a Hoare formula to the general reachability formula:

$$< P \rightsquigarrow K \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < K \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

from which the Hoare triple $\{A\}\ P\ \{B\}$ follows as a special case by applying the constructor substitution $\{K \mapsto done\}$.

Hoare Logic and Reachability Formulas

For the case of a Hoare Logic formula about an IMPL program P, which in traditional notation would be specified as a Hoare triple $\{A\}\ P\ \{B\}$, we will write a reachability formula where A and B are pattern predicates of the form:

$$< P \rightsquigarrow done \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < done \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

where $\vec{x} = x_1, \ldots, x_n$ are the program variables in P, and $\vec{x} \mapsto \vec{X}$ (similar for $\vec{x} \mapsto \vec{X'}$) abbreviates the $VStore$ fragment: $x_1 \mapsto X_1 * \ldots * x_n \mapsto X_n$. It will always be useful to generalize such a Hoare formula to the general reachability formula:

$$< P \rightsquigarrow K \mid TS \ \& \ \vec{x} \mapsto \vec{X} * VS > \mid \varphi \rightarrow^{\circledast} < K \mid TS \ \& \ \vec{x} \mapsto \vec{X'} * VS > \mid \psi$$

from which the Hoare triple $\{A\}\ P\ \{B\}$ follows as a special case by applying the constructor substitution $\{K \mapsto done\}$.

Generalize and Conquer!

Acknowledgements

The continuation-based semantics of IMPL presented in this lecture has been developed in joint work with Michael Abir. A more detailed document containing all the details of IMPL continuation semantics and verification of IMPL program properties is in preparation.