Continuation Semantics and Program Verification for the IMPL Language

Micky Abir and José Meseguer
CS Department, University of Illinois at Urbana-Champaign

Abstract

These notes assume a reader already familiar with the basic concepts of rewriting logic and reachability logic, but not yet familiar with either the formal semantics of programming language or the deductive verification of imperative programs. The IMPL programming language, which supports computation with both numbers and lists, is used as a simple basis through which the reader can become more easily familiar with these two areas and gain an initial experience in specifying properties of imperative programs and verifying then with a proof assistant. Main highlights include: (i) the IMPL syntax, (ii) the formal semantics of IMPL in a continuation-based style, (iii) the specification of reachability logic properties of IMPL programs and its verification with the assistance of the IMPL Prover, obtained as an instantiation of the theory-generic constructor-based reachability logic Prover, and (iv) a proof methodology to reason about program properties in a modular way and arrive at effective proof plans before invoking the help of the IMPL Prover.

1 Introduction

The main goal of the present notes is to provide a tutorial introduction to program verification for programs in an imperative programming language within the rewriting logic semantic framework [2, 3], using constructor-based reachability logic [9] and its theory-generic prover as the logic of programs to express and prove program properties. Therefore, the conceptual approach taken is just an instance of the general approach in the Rewriting Logic Semantics project [4, 8, 5], which makes all program verification in a language $\mathcal{L}$ *semantics-based*, that is, based on $\mathcal{L}$'s formal semantics as a rewrite theory $\mathcal{R}_{\mathcal{L}}$. One important advantage of this approach is that program logics and their supporting tools, including model checkers, theorem provers, and static analysis tools, become *language-generic* and can therefore be reused for many languages, as opposed to the more conventional language-specific approach of crafting a program logic and its tools for a specific programming language $\mathcal{L}$.

That reachability logic is a very effective *language-generic* logic of programs has been amply demonstrated by researchers within the K-approach to program-

ming languages, e.g., [6, 10, 11]. Researchers within the K approach have made enormously important contributions to the rewriting logic semantics project, and have brought its ideas in intimate contact with actual software practice. What the work in [9] on *constructor-based* reachability logic adds to those previous contributions in the area of generic logics of programs is to show how a careful choice of logic primitives that takes into account the algebraic properties of the given rewrite theory $\mathcal{R}$ on which we are reasoning can:

1. Make reachabilty logic not only *language-generic*, i.e., applicable to any programming language $\mathcal{L}$, but also *rewrite-theory-generic*, i.e., applicable to any rewrite theory $\mathcal{R}$ under very general conditions. The point is that $\mathcal{R}$ may in some cases be a theory $\mathcal{R}_{\mathcal{L}}$ specifying the semantics of a programming language $\mathcal{L}$, but in many other cases $\mathcal{R}$ may instead formally specify a *distributed system*, such as a browser, a network protocol, or a cloud-based data storage system.

2. Support powerful *symbolic reasoning* methods in reachability logic theorem proving, thus increasing automation and helping the user focus the proof effort at a higher level of abstraction.

Within the semantics context just described, what these notes address is an eminently practical need: how can a *practitioner* familiar with: (a) the usual features of imperative programming languages, and (b) the basic concepts of rewriting logic and of constructor-based reachability logic (which we do not review in these notes: see [2, 3], and [9] for very detailed introductions), but *not yet* familiar with: (c) the formal semantics of programming languages, and (d) the deductive verification of programs in an imperative language, be helped, so as to relatively quickly become familiar with both these areas and be able to verify programs on his/her own.

Since reachability logic properties are expressed in terms of the chosen rewrite theory $\mathcal{R}$, the first order of business is to make such a practitioner familiar with how one defines a formal semantics $\mathcal{R}_{\mathcal{L}}$ for a programming language $\mathcal{L}$. We address this need by explaining in detail how a *continuation style formal semantics* $\mathcal{R}_{IMPL}$ can be defined for a simple, yet interesting, programming language IMPL supporting both natural numbers and list data structures. Specifically, we define the syntax of IMPL in Section 2, and its continuation style semantics $\mathcal{R}_{IMPL}$ in Section 3. With these notions already understood, and a concrete and not complicated language like IMPL ready to serve as a testing ground to get introduced to the basic notions of program verification, everything is in principle ready to start proving properties of programs. But there is a catch. Even with the best of tools, deductive program verification requieres *careful thought* to understand in depth the program one is interested in and its properties. Only after such careful thought is it meaningful to use a theorem prover as a proof assistant to verify a program. After explaining the basic facts about how the theory-generic constructor-based reachability logic theorem prover described in [9] becomes an IMPL Prover when instantiated with the semantics $\mathcal{R}_{IMPL}$ of IMPL in Section 4, we provide in Section 5 a tutorial

on how to arrive at a careful proof plan for verifying a program *before* invoking the IMPL Prover. Specifically, we present a *compositional proof methodology* that can help a practitioner reason about how to decompose the proof of some desired reachability properties of a program P —including Hoare Logic properties [1], since, as we explain, Hoare logic is naturally embedded as a sublogic of reachability logic. We then illustrate with some concrete IMPL examples how to effectively apply such a methodology to verify specific programs.

For completeness and ease of reference, we include the equations and rewrite rules of the IMPL continuation semantics $\mathcal{R}_{IMPL}$ in Appendix A.

2 IMPL Syntax

The IMPL language, which stands for **IMP** + **L**ists, is a simple imperative language with syntax that resembles C, with the addition of a built-in list type.

2.1 Identifiers

Currently, IMPL supports a small set of predefined basic identifiers, as well as an arbitrary number of composite identifiers built from the basic ones. They are defined as follows:

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

We use a small set of basic identifiers for simplicity. However, an arbitrary number of identifiers can be built from these by means of the comma operator. For example, x, and $x,,$ are both valid identifiers in IMPL.

2.2 Data

IMPL uses three types of built-in data, which are $Bool$, Nat, and $List$. For each of these three kinds of data we explain below the syntax of their *data values*. In the algebraic data type terminology, data values are data expressions built using only *data constructors*. Of course, these data values will also have a number of additional boolean, arithmetic, or list operations. These and their syntax will be explained later.

The simplest type of data is $Bool$, which is defined as expected.

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

As further explained later, the data values of type Nat are defined using three constructors 0, 1 and +, where + is an associative-commutative operation with 0 as its identity element. These constructors support the idea of "counting with one's fingers," where 1 is viewed as a single "finger" and 0 as "no fingers." The syntax is as follows:

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

Note that, thanks to associativity, parentheses are not required around + expressions, yet this causes no semantic ambiguity. Examples of *Nat* data expressions include 0, 1, and 1 + 1 + 1.

As further explained later, the data type *List* of lists contains *Nat* as a subtype and has two constructors, *nil* and an associative list concatenation constructor _$_. Therefore, its syntax is as follows:

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

Again, thanks to associativity, parentheses are not required around $ expressions, yet this causes no semantic ambiguity. Examples of *List* data expressions include 1, 1 + 1 $ 0 $ 1, and *nil*.

2.3 Expressions

We can divide IMPL expressions into three categories: arithmetic, list, and boolean expressions. Arithmetic expressions, or *AExp*, are defined as a subset of standard arithmetic. However, to avoid confusion between the syntax of IMPL and that of the arithmetic operations in its corresponding algebraic data type, a colon ':' is added after each arithmetic operator.

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

We can think of *AExp* as expressions whose evaluation will result in a natural number.

List expressions, or *LExp* are defined as follows:

$$LExp ::= Id \mid List \mid LExp \ \mathbf{\$:} \ LExp \mid \mathbf{first}(LExp) \mid \mathbf{rest}(LExp) \mid \mathbf{last}(LExp) \mid \mathbf{prior}(LExp)$$

Since many algorithms for lists require operations for accessing lists and combining lists, we provide the list concatenation operation, **$:**, **first**, **rest**, **last**, and **prior**. Intuitively, **first** (resp. **last**) of a list L gives the empty list if L is empty, or it gives the first (resp. last) element of the list. Similarly, **rest** (resp. **prior**) either returns the empty list if L is empty, or returns L excluding the first element (resp. the last element).

Boolean expressions, or *BExp*, allow conjunction and negation of other such expressions as well as < comparison between arithmetic expressions and an emptiness test for lists expressions.

$$\begin{aligned} BExp ::= Bool \mid !\, BExp \mid AExp \ <: \ AExp \mid AExp \ <=: \ AExp \\ \mid BExp \ \mathbf{and} \ BExp \mid \mathbf{empty}(LExp) \end{aligned}$$

Similarly, we can think of $BExp$ as expressions that evaluate to the booleans $true$ or $false$. We can use the negation operation, !, and the *and* operation, **and**, to build any other boolean operations such as *or*, *xor*, etc.

Likewise, IMPL provides two arithmetic comparison operations, <: and <=:, which can be used to build up equality check operations and other comparison operations.

We currently only support one boolean operation on lists, the **empty** boolean predicate, which is sufficient for our present purposes. Other list predicates could either be added as basic predicates in a language extension, or could be defined as subroutines within IMPL.

2.4 Statements

Statements are separated into statement concatenation, brackets, variable assignment, and control flow.

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

The empty statement is denoted by two curly brackets, which intuitively can be thought of as a no-op or "skip." Statement concatenation does not rely on a separation character, such as the use of semicolons in C, but is expressed by empty syntax juxtaposition: $Stmt\ Stmt$.

In IMPL, we make a distinction between assignments of arithmetic expressions and of list expressions. Arithmetic assignment uses the standard equals character, whereas a list assignment requires the letter l following the equals character.

We define two types of control flows in IMPL, namely, loops and conditional branches. For loops, we use the standard *while* loop syntax, and for branching we use the conventional $if - else$ syntax.

3 IMPL Continuation Semantics

IMPL is a typed language with variables, implementing both unbounded natural numbers and lists of natural numbers as types. We describe the semantics of IMPL as a continuation-style semantics specified as a rewrite theory

$$\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$$

Choosing a continuation-style semantics for $\mathcal{R}_{IMPL}$ is just one possibility, yet a practical, efficient, and flexible one. As shown in the paper [8], many other styles of defining the operational semantics of a programming language such as, for example, small- and big-step operational semantics, reduction semantics, MSOS, or CHAM semantics could be chosen, so that $\mathcal{R}_{IMPL}$ would then be the rewrite theory defining the semantics of IMPL in that chosen style. In that sense, rewriting logic does not impose any limitation of the choice of different styles of defining a language's semantics. The signature Σ_{IMPL} specifies the grammar of IMPL, just as we have specified it in Section 2, as well as extra operators needed to describe the *states* of the IMPL computations.

In our chosen continuation semantics style, states of the computation are pairs $< \mathit{Continuation} \mid \mathit{Store} >$, where *Continuation* represents "the rest of the program" that remains to be executed, and *Store* is the current store, mapping program variables to their current values. In an initial state of program execution, the first component will be a continuation of the form $P \rightsquigarrow \mathit{done}$, where P is the IMP program P to be executed, and *done* is the continuation where nothing remains to be executed, so that execution terminates. The second second component of the initial computation state will be the initial store. However, $P \rightsquigarrow \mathit{done}$ will be immediately transformed, before program execution begins, into a program *continuation*, i.e., into a sequence of elementary tasks of form:

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

which are then executed one at a time from left to right by the semantic rules R_{IMPL}. However, since some of the tasks T_i may correspond to the evaluation of while loops, some tasks may be recursive, so that when a recursive task T_i reaches the top of the continuation, the equations in E_{IMPL} and the rules in R_{IMPL} may transform such a recursive task T_i into a similar task after one loop iteration. As we shall see, the rewrite rules R_{IMPL}, which all have the general form:

$$< \mathit{Continuation} \mid \mathit{Store} > \rightarrow < \mathit{Continuation}' \mid \mathit{Store}' >$$

always select the "next program evaluation step" T_1 at the top of the current continuation, *Continuation*, as the task to be executed, whereas the rest of the continuation in *Continuation*, that is, $T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1}\mathit{done}$ is what remains to be executed after task T_1. As the result of the application of a rewrite rule of the above form, we reach a new continuation-store pair of the form $< \mathit{Continuation}' \mid \mathit{Store}' >$ corresponding to the new state and new list of pending tasks after the execution of T_1 in *Continuation* in the current *Store*.

3.1 From Programs to Continuations

But how is the initial continuation $P \rightsquigarrow \mathit{done}$ transformed into a corresponding initial list of tasks $T_1 \rightsquigarrow T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1} \rightsquigarrow \mathit{done}$? This is achieved be reducing $P \rightsquigarrow \mathit{done}$ to canonical form with the following subset of equations in E_{IMPL}:

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

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

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

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

$$!BE \rightsquigarrow K\ =\ BE \rightsquigarrow ! \rightsquigarrow K$$
$$BE_1\ \textbf{and}\ BE_2 \rightsquigarrow K\ =\ BE_1 \rightsquigarrow \textbf{and}(BE_2) \rightsquigarrow K$$

$$LE_1\ \textbf{\$:}\ LE_2 \rightsquigarrow K\ =\ (LE_1, LE_2) \rightsquigarrow \textbf{\$:} \rightsquigarrow K$$
$$\textbf{first}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{first} \rightsquigarrow K$$
$$\textbf{rest}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{rest} \rightsquigarrow K$$
$$\textbf{last}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{last} \rightsquigarrow K$$
$$\textbf{prior}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{prior} \rightsquigarrow K$$
$$\textbf{empty}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{empty} \rightsquigarrow K$$

That is, the initial continuation $T_1 \rightsquigarrow T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1} \rightsquigarrow \mathit{done}$ is just the canonical form of $P \rightsquigarrow \mathit{done}$ under the above equations. Note that what is happening is that the above equations transform *program fragments* into 0, 1, or more corresponding *tasks* in the continuation, which have an internal syntax

different from the program syntax. For example, the equation

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

transforms the program-level if-statement **if** (B) S **else** S' into its task representation $B \rightsquigarrow \textbf{if}(S, S')$, where now the first task to be executed is the evaluation of the statement's boolean expression B. But the task of evaluating boolean expression B will itself be transformed by some of the above equations into a sequence of more elementary tasks by the equations in the last three groups of equations above, which handle, respectively, the transformation into elementary tasks for future evaluation of arithmetic, boolean, and list program expressions.

3.2 Continuations

As mentioned above, the initial state of a program evaluation will be a pair

$$< P \rightsquigarrow \mathit{done} \mid \mathit{InitStore} >$$

where P is the program and $InitStore$ is the initial store. Before any rules in R_{IMPL}, which define the execution semantics of IMPL, are applied, as explained in Section 3.1, the initial continuation $P \rightsquigarrow \mathit{done}$ is transformed into a sequence of elementary tasks, which are the canonical form of $P \rightsquigarrow \mathit{done}$ by the subset of equations of E_{IMPL} described in Section 3.1. With this information, we can be more precise about the general form of the rewrite rules in R_{IMPL}. They al have the form:

$$< \mathit{ContTask} \rightsquigarrow \mathit{Continuation} \mid \mathit{Store} > \rightarrow < \mathit{Continuation}' \mid \mathit{Store}' >$$

That is, any continuation in canonical form will always be a sequence of tasks, where $ContTask$ is the first task, at the top of the task sequence, to be executed by some rule in R_{IMPL}.

Therefore, we can define continuations as:

$$\mathit{Continuation} ::= \mathit{done} \mid \mathit{ContTask} \rightsquigarrow \mathit{Continuation}$$

where the different kinds of possible $ContTask$s are the ones we have already encountered in the righthand sides of the subset of equations of E_{IMPL} described in Section 3.1.

3.3 Stores

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

A $VStore$ (for *Value Store*), is defined as a finite map sending each Id to either a Nat or a $List$. The mapping of an identifier x to either a Nat or $List$

data value v is described as a pair $x \mapsto v$ using the maps-to pair constructor $\_ \mapsto \_$. A $VStore$ is a finite function, represented set-theoretically as a set of such pairs built up by means of the associative-commutative union operator $\_ * \_$. Therefore, if $VStore_1$ and $VStore_2$ are two stores with disjoint identifier domains, then $VStore_1 * VStore_2$ is the $VStore$ specifying their disjoint union.

For example, the following is a valid state:

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

Since a $VStore$ can contain both natural and list values, in general it is untyped. To avoid this problem, we define a new kind of store, $TStore$ (for *Type Store*), that records type information. The structure of a $TStore$ is analogous to that of a $VStore$, except that now an identifier is not mapped to a data value but to a $Type$. Here, $TNat$ represents the type of natural numbers and $TList$ represents the type of lists. For the previous $VStore$ example, its corresponding $TStore$ would look as follows:

$$x \mapsto TNat * y \mapsto TList$$

Finally, we define the full store $Store$ as a pair $TStore$ & $VStore$ built up with the pairing constructor $\_\&\_$, where $TStore$ is the type store associated to $VStore$. Therefore, our example full store would look as follows:

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

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

Finally, a state of an IMPL computation will be continuation store, or, $ContStore$, which, as already mentioned, is defined as a pair consisting of a continuation and a store as follows:

$$ContStore ::= < Continuation \mid Store >$$

3.4 Semantic Rules for IMPL

Now that all the components of a continuation store have been explained, we are ready to present the rewrite rules in R_{IMPL} describing the elementary steps of program execution in IMPL. We will also explain a few additional equations in E_{IMPL} not yet presented, which can be better understood in conjunction with some of the rules in R_{IMPL}. We present the most representative rules in R_{IMPL}. A full specification of the entires sets of equations E_{IMPL} and rules R_{IMPL} can be found in Appendix A.

3.4.1 Variable Update and Variable Lookup Semantic Rules

We present the rules for update and lookup of variables of type $TNat$. The corresponding rules for variables of type $TList$ are entirely similar.

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

3.4.2 Arithmetic, Boolean, and List Data Type Rules

We present and explain below a representative subset of the rewrite rules that perform elementary operations for natural numbers and for booleans. The corresponding rules for performing elementary operations on lists are entirely similar. The operation symbols on the righthand sides, like $+$, $*$ and so on, are performed in the associated data types for naturals, booleans and lists. They execute the corresponding program-level operations $+$:, $*$: and so on.

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

The key idea in all of these rules is that data values for (some of) the subexpressions below the top operator in a natural or boolean expression have already been evaluated to their corresponding values, so that now the result of applying some elementary operation to those values can be computed in the underlying data type. For example, the 2-tuple (I_1, I_2) in the first and second rules will contain the results of having previously evaluated the arithmetic subexpressions AE_1 and AE_2 in some arithmetic expression, $AE_1 +: AE_2$ (resp. $AE_1 *: AE_2$). However, for some other operations, such as boolean conjunction, it may sometimes be unnecessary to evaluate both arguments of an expression to their resulting values: when the first argument of a conjunction expression evaluates to *false*, its second argument, i.e., the boolean expression BE, needs no evaluation. But if the first argument evaluates to *true*, BE must be evaluated.

3.4.3 Tuple Continuation Equations

Except for boolean conjunction, all other binary operations in expressions follow the same pattern. Such binary operations take either two natural or two list argument expressions. Since the approach for natural and list subexpressions is similar, let us consider the case of an expression of the form AE_1 *op*: AE_2, with AE_1, AE_2 arithmetic expressions. A continuation of the form AE_1 *op*: $AE_2 \rightsquigarrow K$ is transformed by one of the E_{IMPL} equations listed in Section 3.1 into a continuation of the form $(AE_1, AE_2) \rightsquigarrow op: \rightsquigarrow K$. But there is a mystery that still needs to be clarified. How does the 2-tuple of expressions (AE_1, AE_2) get evaluated to their corresponding values (I_1, I_2) in the current store? And how is that done following a left-to-right evaluation order? This is where tuple continuation equations come in. There are three for 2-tuples of arithmetic expressions, and three completely similar ones for 2-tuples of list expressions. The three for 2-tuples of arithmetic expressions are:

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

That is, we first pull out of the 2-tuple its first expression AE_1, put it at the top of the continuation, and replace its "hole" by the place holder #. This will usually trigger further application to AE_1 of the E_{IMPL} equations listed in Section 3.1 (plus the above 2-tuple equations) . After AE_1 gets eventually evaluated to its value I_1 by the semantic rules, then we put I_1 into the first component of the tuple, pull AE_2 to the front, and put the place holder # in the second component. Finally, when AE_2 gets eventually evaluated to I_2, we put I_2 into the second component to get the 2-tuple of values (I_1, I_2), to which the rule in Section 3.4.2 handling the evaluation of *op*: will apply.

3.4.4 Branching Semantic Rules

The only remaining rules are the ones for branching on a condition, which are applied after its boolean condition B gets evaluated to either *true* or *false*.

$$
\begin{aligned}
&< true \rightsquigarrow \mathbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S \rightsquigarrow K \mid St > \\
&< false \rightsquigarrow \mathbf{if}(S, S') \rightsquigarrow K \mid St > \rightarrow < S' \rightsquigarrow K \mid St >
\end{aligned}
$$

In summary, 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 way in which the original continuation $P \rightsquigarrow \mathit{done}$ gets transformed by the equations E_{IMPL} into a sequence of simple elementary tasks. The only slightly more subtle equation is the recursive one for while loops, $\textbf{while}\ (BE)\ \{S\} \rightsquigarrow K \ = \ BE \rightsquigarrow \textbf{if}(\{S\ \textbf{while}\ (BE)\ \{S\}\}\ ,\ \{\}) \rightsquigarrow K$, which, after BE gets evaluated to either *true* or *false*, triggers the application of one of the two branching rules above, may apply again to the first component of the $\textbf{if}(\{S\ \textbf{while}\ (BE)\ \{S\}\}\ ,\ \{\})$ pair, and marks the essential difference between always terminating programs without loops, and a Turing complete language like IMPL.

3.5 Abstract versus Fine Grained Continuation Semantics

The continuation semantics of IMPL jut specified by the rewrite theory $\mathcal{R}_{IMPL} = (\Sigma_{IMPL}, E_{IMPL} \cup B, R_{IMPL})$, whose equations E_{IMPL} and rules R_{IMPL} are fully described in Appendix A, makes a distinction between *execution steps* performed by the rules R_{IMPL}, and *syntax manipulation steps* performed by the equations E_{IMPL}, which transform the abstract syntax tree of a program P into its continuation representation as a sequence of tasks $T_1 \rightsquigarrow T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1} \rightsquigarrow \mathit{done}$. In this sense, the semantics of $\mathcal{R}_{IMPL}$ *abstracts away* the syntactic manipulations of the program P, since all intermediate manipulation steps are equivalent in the canonical reachability model of $\mathcal{R}_{IMPL}$ to the canonical form $T_1 \rightsquigarrow T_2 \rightsquigarrow \ldots T_n \rightsquigarrow T_{n+1} \rightsquigarrow \mathit{done}$ of the original $P \rightsquigarrow \mathit{done}$. This means that *considerably fewer steps count as computation steps*: syntactic manipulations are there but they become *invisible*. It is of course possible to *make them visible* is a more *fine grained* continuation semantics. How so? By *turning equations into rules*. All equations in E_{IMPL} are of the form $k = k'$, with k and k' terms of sort *Continuation*. So we could just transform them into rules $k \rightarrow k'$. However, one of the highly desirable properties of the rules R_{IMPL} is that they are *topmost*, i.e., they rewrite the *entire state*, as a continuation-store pair. Can we achieve that when transforming the equations E_{IMPL} into rules? Very easily: we just transform each equation $k = k'$ into the rule $< k \mid St > \rightarrow < k' \mid St' >$. For example, the equation $AE_1 +: AE_2 \rightsquigarrow K \ = \ (AE_1, AE_2) \rightsquigarrow +: \rightsquigarrow K$ becomes the rule

$$< AE_1 +: AE_2 \rightsquigarrow K \mid St > \rightarrow < (AE_1, AE_2) \rightsquigarrow +: \rightsquigarrow K \mid St >$$

Let $\vec{E}^{\bullet}_{IMPL}$ denote the set of rules thus obtained. Then, the *fine-grained continuation semantics* of IMPL is the rewrite theory $\mathcal{R}^{FG}_{IMPL} = (\Sigma_{IMPL}, B, \vec{E}^{\bullet}_{IMPL} \cup R_{IMPL})$. Each version has advantages and disadvantages. $\mathcal{R}_{IMPL}$ is of course more abstract, and by having *fewer computation steps* allows shorter proofs of reachability properties, since the **Step** inference rule only uses the rules in R_{IMPL} and the equations E_{IMPL} allow us to "fast forward" to states with continuations in canonical form. On the other hand, $\mathcal{R}^{FG}_{IMPL}$ is in some ways simpler and *more detailed*, and has the useful advantage of allowing a *simpler expression of reachability properties*. How so? Because the continuation part in the precondition of such properties can always be written in the form $P \rightsquigarrow \mathit{done}$,

which is easier to write and undrestad than its associated continuation sequence. In fact, in what follows and in using the IMPL prover we will use the Maude specification of $\mathcal{R}^{FG}_{IMPL}$ instead than that of $\mathcal{R}_{IMPL}$. As we shall see, at the expense of having to perform more proof steps, this choice has the advantage of leading to quite easy to understand reachability formulas.

4 The IMPL Prover

How can we prove properties about IMPL programs with the assistance of a theorem prover? In usual practice, a different theorem prover must be developed for each different language. Developing such a prover is typically a several man-year effort. Furthermore, it is also often an *ad-hoc* effort. How so? Because the *semantics* of the given language is only *implicit* in the code of the prover, since such a code is based on the subjective *understanding* that the prover implementers have about the programming language. So what? So such an ad-hoc prover may give *wrong* answers, and declare a program correct when it is not so. This is not a hypothetical case, but a real problem in practice.

There is, furthermore, a second related problem. Which *logic* shall we use to specify *program properties* and prove them with a prover's assistance? The traditional answer has been to use some flavor of Hoare logic. But there are as many Hoare logics as programming languages. Such logics can be quite complex (for example, some Java provers need about 70 Hoare logic inference rules), hard to get right and prove correct (sometimes this is not even done), and require a large implementation effort. Furthermore, a user proving Java programs may need to master 70 quite complex Hoare logic inference rules just to be able to specify Java program properties.

So, what can be done? A much better approach to proving program properties could be achieved if we could develop provers and express program properties in a *language-generic* manner. That is, the theorem prover in question, call it TP, would be a *language-generic* theorem prover. Assuming it is written in Maude, it would not be a fixed rewrite theory, but a *rewrite theory transformer* of the form:

$$TP : \mathcal{R}_{\mathcal{L}} \mapsto TP(\mathcal{R}_{\mathcal{L}})$$

That is, the generic theorem prover TP, when instantiated with a rewriting logic semantics $\mathcal{R}_{\mathcal{L}}$ for a programming language $\mathcal{L}$, yields a specific theorem prover $TP(\mathcal{R}_{\mathcal{L}})$ for $\mathcal{L}$ which is *directly based* on its formal semantics $\mathcal{R}_{\mathcal{L}}$. Of course, once we get a theorem prover $TP(\mathcal{R}_{\mathcal{L}})$ for language $\mathcal{L}$, we want to use it to prove properties about programs in $\mathcal{L}$. But how can we express such properties? In this language-generic approach to program proving, we obviously need a logic of programs that, unlike traditional Hoare logic, is itself also *language-generic*.

Can all this be done? The answer is yes! It has in fact been done several times as part of the *rewriting logic semantics project* [4, 8, 5]. First of all, Roşu, Stefanescu, and other members of the K Team showed that reachability logic can be used as a *language-generic* logic of programs applicable to a wide variety of programming languages [6, 10, 11]. And, more recently, Skeirik, Stefanescu, and

Meseguer have developed constructor-based reachability logic [9] as a property logic that is not only *language-generic*, but also *rewrite-theory-generic*. That is, it can be used to prove properties not only of programs in any programming language, but also of concurrent systems specified as rewrite theories. So, we can choose the constructor-based reachability logic prover described in [9] as our language-generic theorem prover $TP : \mathcal{R}_{\mathcal{L}} \mapsto TP(\mathcal{R}_{\mathcal{L}})$, and specialize it to the IMPL language as the prover $TP(\mathcal{R}_{IMPL})$. This is exactly the approach taken in this section.

Since Hoare logic properties can be easily expressed as reachability logic properties, the logic of our IMPL prover $TP(\mathcal{R}_{IMPL})$ can express any such Hoare logic properties. We therefore show in what follows how to construct a Hoare triple for an IMPL program, transform this triple into a reachability logic goal, and specify and prove this goal in our IMPL prover.

Consider the following IMPL program $\mathbf{P}$, which we will call *migrate*, that moves the elements of a list x into the list y, one element per iteration, with initial value store VS_0.

$$VS_0 := x \mapsto X \ * \ y \mapsto Y \ * \ z \mapsto Z$$
$$\mathbf{P} := \textbf{while } (!empty(x)) \ \{y =_l y \ \$: first(x) \ ; \ x =_l rest(x) \ ; \}$$

We can say the value store after the loop, VS, is

$$VS := x \mapsto X' \ * \ y \mapsto Y' \ * \ z \mapsto Z'$$

where every variable is of type $TList$ in the type store TS.

To express this as a Hoare triple over the rewrite theory $\mathcal{R}_{IMPL}$ of IMPL, we first consider its precondition and postcondition. To keep it simple, we will use the precondition formula $\varphi_0 := (Z) = (Y \$ X)$, specifying that Y is a sublist of Z starting at the head, and X is the remaining sublist of Z. We will also use the postcondition formula $\varphi := (Z) = (Y' \$ X')$, which, as we will see later, makes this relationship our *loop invariant*, with Z as its data parameter.

In order to express this Hoare triple by pattern predicate formulas describing IMPL computation states, we replace $\mathbf{P}$ by the continuation $\mathbf{P} \rightsquigarrow done$, which allows $\mathcal{R}^{FG}_{IMPL}$ to execute P until completion, denoted by the continuation *done*. To put it all together, we have the following Hoare triple:

$$\{< \mathbf{P} \rightsquigarrow done \mid TS \ \& \ VS_0 > \ \mid \varphi_0\} \ \mathcal{R}_{\mathcal{L}} \ \{< K \mid TS \ \& \ VS > \ \mid \varphi\}$$

where K is a variable of sort *Continuation*. Transforming this Hoare triple into a reachability logic goal is straightforward. The only extra thing we need to do is to explicitly reflect that —since we are interpreting a Hoare triple is interpreted only in *terminating* states, that the corresponding reachability formula's *midcondition* will *actually be a postcondition*. Since for IMPL any

terminating state is a pair whose continuation part is the *done* continuation, and whose store part is the store St at the end of program execution, all we need to do is to *explicitly constrain the postcondition in the Hoare triple* to only describe a terminating states, which is easy by just *instantiating* the continuation variable K in the Hoare triple's postcondition to *done*. In this way, the above Hoare triple can be expressed as the following reachability logic goal:

$$< \mathbf{P} \rightsquigarrow \mathit{done} \mid TS \,\&\, VS_0 > \mid \varphi_0 \;\; \rightarrow^{\circledast} \;\; < \mathit{done} \mid TS \,\&\, VS > \mid \varphi$$

However, as explained in Section 5.1.3, it is *easier* to prove this goal by *first generalizing it* to the most general setting of beginning with a continuation $\mathbf{P} \rightsquigarrow K$ and ending with a continuation K. That is, we place P in the context of some further computation K to be done afterwards, with $K = \mathit{done}$ as a special case. This is called in Section 5.1.3 a *Generalize and Conquer* method: proving something more general is often simpler. So we will prove the more general goal:

$$< \mathbf{P} \rightsquigarrow K \mid TS \,\&\, VS_0 > \mid \varphi_0 \;\; \rightarrow^{\circledast} \;\; < K \mid TS \,\&\, VS > \mid \varphi$$

The last thing we must do is to specify this goal in our IMPL prover using the syntax of the reachability logic prover. We begin by specifying the set of terminating states from above:

```
(def-term-set (< done | St:Store >) | true .)
```

Next, we need to declare the logical variables that we will use, which are A, I, and X.

```
(declare-vars (X:List) U (X':List) U (Y:List) U (Y':List) U
              (Z:List) U (Z':List) U (K:Continuation) .)
```

Note that these variables are of type $List$ and not of type $TList$, since these are logical variables and not program variables.

To add our reachability claim, we add it as a proof goal with a descriptive label *migrate*.

```
(add-goal migrate :
 (< while (! empty(x)) { y =l y $: first(x) ; x =l rest(x) ; } ~> K
     | (x |-> TList * y |-> TList * z |-> TList)
     & (x |-> X * y |-> Y * z |-> Z) >)
     | (Z) = (Y $ X)
     =>
     (< K | (x |-> TList * y |-> TList * z |-> TList)
     & (x |-> X' * y |-> Y' * z |-> Z') >)
     | (Z) = (Y' $ X') .)
```

Once the proof goal is specified, we are ready to start and carry out the proof, which we will see how to do in section 5.

5 Proof Methodology

Although the IMPL Prover provides an interface allowing us to prove IMPL programs by giving proof commands, not all such proof commands will be equally successful. Before giving any proof commands, a deeper understanding of the problem at hand will be invaluable in proving interesting properties. This deeper understanding can be both developed and aided by following a specific *proof methodology*. Such a proof methodology approaches the verification of a program P in two steps:

1. **Proof Design**. Given a program P and a desired property of P, viewed as either a Hoare triple or as a reachability formula (our *main goal*), we carefully *design* a collection of *auxiliary properties* about P or about *fragments* of P that we can use as useful *lemmas* in proving P. In doing so, we try to reason *compositionally*. That is, by *decomposing* a program into the smaller subprograms it is composed of. In this way, we can likewise *decompose* the program proving task into smaller subtasks. The final result is a *proof plan*, where the proof of our main goal is decomposed into simpler proofs of smaller subgoals.

2. **Proof Implementation**. Once we have designed a proof plan for proving a given property of our program P, we *implement* such a plan by giving commands to the IMPL Prover. Such an implementation may often take the form of *first proving some reachability formulas as auxiliary lemmas*, and then using such already proved reachability formulas as *axioms* in proving the main goal.

Before considering concrete examples of proofs of IMPL programs, we present some *general proof methods* that can guide our *proof design* for proving properties of specific programs. After explaining such general proof methods, we illustrate how to apply them to prove specific properties of concrete IMPL programs.

5.1 General Proof Methods

Before discussing general methods, let us introduce some *useful notation* and explain the notion of *pattern subsumption*.

5.1.1 Notation

In a *fine-grained* continuation semantics (see Section 3.5), the patern predicates appearing in reachability formulas for IMPL programs will have the general form[1] $< P \rightsquigarrow \mathit{done} \mid \mathit{st} >| \varphi \rightarrow^{\circledast} < \mathit{done} \mid \mathit{st}' >| \psi$ for a Hoare triple, or, more generally, $< P \rightsquigarrow K \mid \mathit{st} >| \varphi \rightarrow^{\circledast} < K \mid \mathit{st}' >| \psi$ for a reachability formula,

[1] More generally, the midcondition (in specific cases the postcondition) could of course be a disjunction of pattern predicates. However, in the examples we shall consider all midconditions will be single pattern predicates. Since this is a common case, we will assume such single-pattern midconditions in what follows.

where st and st' are specific *store patterns* describing the shape of the pre- and mid-stores.

But this is not concrete enough. How do st and st' typically look like? Let $\vec{x} = x_1, \ldots, x_n$ denote the set of *program variables* ever read or written to by program P. Then st (resp. st') has typically the form:

$$st = (TS \;\&\; \vec{x} \mapsto \vec{X} * VS) \qquad resp. \quad st' = (TS \;\&\; \vec{x} \mapsto \vec{X'} * VS)$$

where $\vec{x} \mapsto \vec{X}$ (resp. $\vec{x} \mapsto \vec{X'}$) abbreviates the *VStore* fragment $x_1 \mapsto X_1 * \ldots * x_n \mapsto X_n$ (resp. $x_1 \mapsto X'_1 * \ldots * x_n \mapsto X'_n$).

How about the *parameters* in a reachability formula of the form

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

Obviously, the *continuation variable* K is one of them. Let us assume the very common case where the remaining parameters, let us call them the *data parameters*, are all in fact a *subset* $\vec{Y} \subseteq \vec{X}$ of the set $\vec{X}$ of *logical variables* $\vec{X}$ in the VS pattern $\vec{x} \mapsto \vec{X}$. Unless the subset $\vec{y} \subseteq \vec{x}$ of program variables where the logical variables $\vec{Y}$ are stored are all *read-only* variables, only *some* of the logical variables in $\vec{Y}$ (perhaps none) may also appear in $\vec{X'}$. To allow for this possibility, it may be useful to make explicit the logical variables on which the constrain formulas φ and ψ *may* depend, namely, $\varphi = \varphi(\vec{X})$ and $\psi(\vec{Y}, \vec{X'})$, with $\vec{Y} \subseteq \vec{X}$ and $(\vec{X'} \backslash \vec{Y}) \cap \vec{X} = \emptyset$. To make explicit and emphasize in the notation *which* are the data parameters $\vec{Y}$, we may sometimes write $\varphi(\vec{X})$ in the redundant form $\varphi(\vec{Y}, \vec{X})$.

A second bit of useful notation regards notation for (arithmetic or list or boolean) *expressions* in IMPL. The same way that t denotes a Σ-term or expression in a given signature Σ of operators, let $t{:}(\vec{x})$ denote an IMPL expression mentioning program variables $\vec{x}$. The point of this notation is that, given a VS pattern $\vec{x} \mapsto \vec{X}$, we can associate to the IMPL expression $t{:}(\vec{x})$ a corresponding Σ_D-expression $t(\vec{X})$, where Σ_D is the signature of the *underlying data type* (combining natural, list and boolean operations) where the IMPL expressions are *evaluated* according to their continuation semantics. For example, if $t{:}(\vec{x})$ is the IMPL expression x *: $(y+{:}\ x)$, given the VS pattern $x \mapsto X * y \mapsto Y$, we obtain the Σ_D-expression $X * (Y + X)$.

5.1.2 Pattern Subsumption

The theory-generic reachability logic prover *RLP* of which the IMPL Prover is the instantiation $RPL(\mathcal{R}^{FG}_{IMPL})$ supports the `(subsumed Pattern =< Pattern' .)` command. To prove that the set of states denoted by `Pattern` is contained in the set of states denoted by `Pattern'`. But how is such command executed by *RLP*? By a symbolic evaluation method called *pattern subsumption*. Given constructor pattern formulas $u \mid \varphi$ and $v \mid \psi$, we say that $u \mid \varphi$ is *subsumed* by

$v \mid \psi$, denoted $u \mid \varphi \sqsubseteq v \mid \psi$, iff (by definition) there exists a substitution α such that: (i) $u =_B v\alpha$, and (ii) $\mathcal{T}_{\Sigma/E\cup B} \models \varphi \Rightarrow (\psi\alpha)$, where $(\Sigma, E \cup B)$ is the equational theory of the given rewrite theory $\mathcal{R}$ where we are reasoning. That is, the implication $\varphi \Rightarrow (\psi\alpha)$ must be shown to be an *inductive theorem* in the initial algebra $\mathcal{T}_{\Sigma/E\cup B}$ As shown in [9], whenever $u \mid \varphi \sqsubseteq v \mid \psi$ holds, then the subset inclusion between sets of states $[\![u \mid \varphi]\!] \subseteq [\![v \mid \psi]\!]$ holds too. In the case of *parametric* formulas with set Y of parameter variables, there is also a notion of *parametric pattern subsumption* defined as a relation $u \mid \varphi \sqsubseteq_Y v \mid \psi$ that holds iff (by definition) there exists a substitution α which leaves Y unchanged, i.e., for each $y \in Y$ $\alpha(y) = y$, and such that: (i) $u =_B v\alpha$, and (ii) $\mathcal{T}_{\Sigma/E\cup B} \models \varphi \Rightarrow (\psi\alpha)$. If $u \mid \varphi \sqsubseteq_Y v \mid \psi$ holds, then for each constructor substitution $\rho \in [Y \to T_\Omega]$ we have a subset inclusion between sets of states $[\![(u \mid \varphi)\rho]\!] \subseteq [\![(v \mid \psi)\rho]\!]$.

5.1.3 Compositional Proof Methods

Before discussing a number of proof methods that will be useful in proving reachability properties of IMPL programs, a few words are in order about the *intellectual status* of the methods to be discussed. The logic, implementation, and correctness of the IMPL Prover does *not* depend on any of the methods discussed below at all. Remember that the IMPL Prover is just the instantiation $RPL(\mathcal{R}^{FG}_{IMPL})$ of the *RLP* constructor-based reachability prover. Therefore, since *RLP* is a *rewrite-theory-generic* prover, its proof methods know nothing about IMPL, loops, statements, stores, continuations, or anything like that: $\mathcal{R}^{FG}_{IMPL}$ is just *one more rewrite theory*. If $RPL(\mathcal{R}^{FG}_{IMPL})$ proves some reachability properties about an IMPL program P, no special knowledge about IMPL has been used. In particular, except for $\mathcal{R}^{FG}_{IMPL}$ itself, no *specific knowledge* about the IMPL semantics is ever used in such proofs. So, what is the intellectual status of the methods to be discussed? As we shall see, the only *properties* on which some of the methods, such as those for proving properties of loops, of conditional statements and of statement compositions, *depend* are general *theory-generic* properties of reachability logic. Other than that, what all methods we present do is to *use language-specific properties* of IMPL's *continuation semantics*, as specified in $\mathcal{R}^{FG}_{IMPL}$ to *guess* a good proof plan. Whether such a proof plan *succeeds* or not when submitted to the IMPL Prover does *not* depend at all on any particular assumptions extraneous to reachability logic, and therefore do not affect in any way the *validity* of the properties that $RPL(\mathcal{R}^{FG}_{IMPL})$ can prove. However, these methods can be enormously useful in practice. Why? Because they can help *guide the user* in *guessing* and arriving at a *proof plan* that will give the $RPL(\mathcal{R}^{FG}_{IMPL})$ Prover a reasonably good chance to prove the program properties in question. In summary, therefore, this section is about how to *guess and find* a good proof plan to prove our desired properties about a given IMPL program P.

Proving Hoare Triples. Now that we have a more detailed notation about reachability formulas for IMPL, we can be more precise about how a Hoare

triple looks like as a reachability formula. It is a formula of the form

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

which may have data parameters $\vec{Y}$. How can we prove it? Easy. Prove instead the formula

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

which in addition to the data parameters $\vec{Y}$ has the continuation parameter variable K. Since, by definition, a valid parametric reachability formula remains valid after substituting any of its parameter variables by a *constructor term*, if we prove the more general formula above, the reachability formula associated to the original Hoare logic triple is also a trivially valid consequence of it, obtained by applying the constructor substitution $\{K \mapsto \mathit{done}\}$. We can call this method of proving Hoare triples:

Generalize and Conquer

Proving Properties of Assignments. Suppose that we have a pattern formula for the precondition of an assignment statement and we would like to make a *correct guess* for the pattern formula of its midcondition. That is, assuming an assignment with an arithmetic expression (the case for list assignments is entirely similar) we would like to resolve the guess:

$$< x_0 = t{:}(x_0, \vec{x}); \rightsquigarrow K \mid TS \;\&\; x_0 \mapsto X_0 * \vec{x} \mapsto \vec{X} * VS >\mid \varphi \rightarrow^{\circledast} \; ??$$

where the arithmetic expression $t{:}(x_0, \vec{x})$ *may* mention the program variable x_0 (in general it need not do so). What can we do? The following is always a correct guess:

$$\begin{aligned} &< x_0 = t{:}(\vec{x}); \rightsquigarrow K \mid TS \;\&\; x_0 \mapsto X_0 * \vec{x} \mapsto \vec{X} * VS >\mid \varphi \\ &\rightarrow^{\circledast} < K \mid TS \;\&\; x_0 \mapsto X_0' * \vec{x} \mapsto \vec{X} * VS >\mid \varphi \wedge X_0' = t(X_0, \vec{X}). \end{aligned}$$

Proving Properties of While Loops. Suppose we have a reachability formula that we wish to prove about a while loop, say,

$$< \textbf{while}\ b{:}(\vec{x})\ \{\mathit{stmt}\} \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid \varphi \rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X}' * VS >\mid \psi$$

where $b{:}(\vec{x})$ is the loop's boolean expression or *guard*, and the statement *stmt* is the loop's *body*. Suppose also that, besides the continuation parameter variable K, $\vec{Y}$ are the data parameters. Giving such a reachability formula to the IMPL prover without any forethought may often be quite ineffective. What can we do? We can consider two ideas. First, it is a basic property of reachability logic that, by definition, a reachability formula $A \rightarrow^{\circledast}_{Y} D$ about a given rewrite theory $\mathcal{R}$, *parametric* on variables Y (indicated by the arrow $\rightarrow^{\circledast}_{Y}$), will always hold, provided that another formula $B \rightarrow^{\circledast}_{Y} C$ with a *more general* precondition

B and a *less general* midcondition C, i.e., such that $A \sqsubseteq_Y B$ and $C \sqsubseteq_Y D$ is valid in $\mathcal{R}$. This can allow us to *shift our ground*: to prove $A \rightarrow^{\circledast}_Y D$. In our case, $A \rightarrow^{\circledast}_Y D$ will be the original loop reachability property, which can be abbreviates as:

$$A[\textbf{while } b\!:\!(\vec{x})\ \{stmt\} \rightsquigarrow K] \rightarrow^{\circledast} D[K]$$

where on both sides we highlight in square brackets the *continuation subexpression* parts of the pattern formulas A and D viewed as formula expressions. But it will be enough to prove the, hopefully easier to prove, formula:

$$B[\textbf{while } b\!:\!(\vec{x})\ \{stmt\} \rightsquigarrow K] \rightarrow^{\circledast} C[K]$$

But how could this second formula be *easier to prove*? The key remark is that loops are a *repetitive computation*. Therefore, if we could *guess* a property I that is *preserved* by executing the loop, and can therefore be called an *invariant* of it, we can then use such a property I to *choose* B and C to be of the form:

$$I[\textbf{while } b\!:\!(\vec{x})\ \{stmt\} \rightsquigarrow K] \rightarrow^{\circledast} I\sigma[K]$$

where $I\sigma$ is a renaming of the *logical variables* in I. This format for our reachability property will "wear on its sleeve" the repetitive nature of the loop, and will greatly increase our chances that reachability logic's **Axiom** rule, which is a "seven league boots" rule designed to *detect* repetitive behavior, will kick in and will allow us to prove our desired loop property. These two insights are combined in the notion of a *loop invariant*, a method based on the following steps:

1. We guess an *invariant* I that: (i) will be true when entering the loop, and (ii) will be preserved by the body subprogram *stmt*. That is, a formula $I(\vec{Y}, \vec{X})$ such that, under the assumption that the loop's guard holds (so that *stmt* will be executed), *would* allows us to prove the goal:[2]

$$\begin{array}{l} < stmt \rightsquigarrow K \mid TS\ \&\ \vec{x} \mapsto \vec{X} * VS >\mid I(\vec{Y}, \vec{X}) \wedge b(\vec{X}) = true \\ \rightarrow^{\circledast} < K \mid TS\ \&\ \vec{x} \mapsto \vec{X'} * VS >\mid I(\vec{Y}, \vec{X'}) \end{array}$$

   Intuitively, $I(\vec{Y}, \vec{X})$ is a property that holds *before stmt* is executed (assuming $b(\vec{X}) = true$), and *after stmt* is executed, thus the name *invariant*. Although some provers have built-in heuristics that can help in guessing I in some cases, there is in general no free lunch: one has to *think* about what *stmt* is doing to see what properties it preserves. *Then*, and only then, can one guess a reasonable invariant. Furthermore, there is no *single* such invariants to be guessed, and the guess may be *context-dependent* (more on this below).

[2] We do not need to actually prove this goal for *stmt*. It is enough to be sure that it could be proved. We are just using this knowledge to *guess* the right invariant for the loop itself.

2. Since a *terminating* execution of the loop (and remember that reachability logic properties are *partial correctness* assertions that only care about terminating executions) will just execute *stmt* a finite number of times until $b{:}(\vec{x})$ becomes false, the invariant I will hold true not only *before* entering the loop, but also *immediately after* exiting it. Furthermore, after exiting the loop in a state of the form, say, $< K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS >$ we *know* about *two* properties that such a state will satisfy, namely, (i) $I(\vec{Y}, \vec{X'})$, and (ii) $b(\vec{X'}) = \mathit{false}$. Therefore, in Hoare logic approaches it is common to use the data constrain $I(\vec{Y}, \vec{X'}) \wedge b(\vec{X'}) = \mathit{false}$ in a loop's postcondition. It would be natural to use the same data constraint in the loop's midcondition in reachability logic. This makes the midcondition *tighter*, but is uses a somewhat more complex formula. For simplicity, in what follows we will use the looser midcondition $I(\vec{Y}, \vec{X'})$, which is sufficient for proving interesting examples; but all we say can be adapted to using instead the tighter midcondition. Therefore, a good *initial guess* for a property to prove about our loop is:

$$< \mathbf{while}\ b{:}(\vec{x})\ \{stmt\} \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid I(\vec{Y}, \vec{X})$$
$$\rightarrow^{\circledast} < K \mid TS \;\&\vec{x} \mapsto \vec{X} * VS >\mid I(\vec{Y}, \vec{X'})$$

3. Our initial guess, however, may not be the last and best choice. Why not? For two reasons: (1) To begin with, the invariant $I(\vec{Y}, \vec{X})$ may be *too general*, what is called a *weak invariant*, so that, even if it is true, it may be too hard for the IMPL prover to prove it because not strong enough assumptions can be made to achieve a proof. (2) Furthermore, remember that the *original formula* we wanted to prove was:

$$< \mathbf{while}\ b{:}(\vec{x})\ \{stmt\} \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid \varphi$$
$$\rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS >\mid \psi$$

   But such original formula and our guess loop invariant may not "fit together" well. For example, $I(\vec{Y}, \vec{X'})$ may be *too general*, and therefore may not allow us to prove that the original midcondition property ψ holds after executing the loop. The problem is that there is *context information* about φ and ψ that may not have yet been taken into account when crafting $I(\vec{Y}, \vec{X})$ and we need to account for it. Reasons (1) and (2) all move us in the same direction. What we should do is to *strengthen the invariant* into a stronger version $I_{str}(\vec{Y}, \vec{X}) = I(\vec{Y}, \vec{X}) \wedge \phi$ which addresses problems (1) and (2).

4. We can then put everything together as follows. To make sure that the strengthened invariant I_{str} and our reachability original goal fit together well, we need to prove the two parametric subsumptions:

$$< \mathbf{while}\ b{:}(\vec{x})\ \{stmt\} \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid \varphi$$
$$\sqsubseteq_{K,\vec{Y}} < \mathbf{while}\ b{:}(\vec{x})\ \{stmt\} \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid I_{str}(\vec{Y}, \vec{X})$$

and

$$< K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS >\mid I_{str}(\vec{Y}, \vec{X'})$$
$$\sqsubseteq_{K,\vec{Y}} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS >\mid \psi$$

using the `(subsumed Pattern =< Pattern' .)` command. If this is the case, as already mentioned based on general properties of reachability logic, we have then *reduced* the proof of our original reachability formula for the loop to proving the strengthened loop invariant:

$$< \textbf{while } b(\vec{x}) \; \{stmt\} \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid I_{str}(\vec{Y}, \vec{X})$$
$$\rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid I_{str}(\vec{Y}, \vec{X'}).$$

All these ideas are illustrated with a concrete example of an IMPL program in Section 5.2 below.

Note that this method *is directly based* on the theory-generic basic property that if a reachability formula holds for a bigger precondition and smaller post-condition, then the original formula also holds. Therefore, its correctness does not depend at all on the particularities of IMPL.

Proving Properties about Conditional Statements. Suppose we want to prove a reachability formula of the form:

$$< \textbf{if } (b(\vec{x})) \; stmt \; \textbf{else} \; stmt' \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid \varphi$$
$$\rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS >\mid \psi$$

What can we do? We can remember the **Split** auxiliary inference rule of Reachability logic [9], that ensures that the validity of a reachability formula of the form $u \mid \varphi \rightarrow^{\circledast} B$ in a rewrite theory $\mathcal{R}$ is equivalent to the validity of the two reachability formulas $u \mid \varphi \wedge \psi_1 \rightarrow^{\circledast} B$ and $u \mid \varphi \wedge \psi_2 \rightarrow^{\circledast} B$, provided ψ_1 and ψ_2 do not have any extra variables besides those of $u \mid \varphi$, and $\mathcal{T}_{\Sigma/E \cup B} \models \psi_1 \vee \psi_2$, where $\mathcal{T}_{\Sigma/E \cup B}$ is the initial algebra of $\mathcal{R}$'s underlying equational theory. How can we apply the **Split** rule to the above goal? Since the equational theory $(\Sigma_D, E_D \cup B_D)$ of the combined algebraic data type containing all the arithmetic, boolean, and list operations needed to evaluate the program expressions of IMPL is *protected* as a subtheory of the equational theory $(\Sigma_{IMPL}, E_{IMPL} \cup B)$ of $\mathcal{R}_{IMPL}$, i.e., we have an isomorphism $\mathcal{T}_{\Sigma_{IMPL}/E_{IMPL} \cup B}|_{\Sigma_D} \cong \mathcal{T}_{\Sigma_D/E_D \cup B_D}$, and we also have $\mathcal{T}_{\Sigma_D/E_D \cup B_D} \models b(\vec{X}) = \mathit{true} \vee b(\vec{X}) = \mathit{false}$, we can apply the **Split** rule of reachability logic to reduce our original goal to the simpler two goals semantically equivalent to it:

$$< stmt \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid \varphi \wedge b(\vec{X}) = \mathit{true}$$
$$\rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS >\mid \psi$$
$$< stmt' \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS >\mid \varphi \wedge b(\vec{X}) = \mathit{false}$$
$$\rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS >\mid \psi.$$

Note that this method *is directly based* on the **Split** rule of reachability logic, which is theory-generic: its correctness does not depend at all on the particularities of IMPL.

The Chain Rule. The following is a *general property of reachability logic* that can also be very useful in proving IMPL programs. Suppose that $A \rightarrow^{\circledast}_{Y} B$ and $B \rightarrow^{\circledast}_{Y} C$ are two reachability formulas about a given rewrite theory $\mathcal{R}$, both *parametric* on variables Y and such that the only variables shared by A and C are exactly the variables Y (notice that we assume that B is a single-pattern formula). Then, if we can prove that $A \rightarrow^{\circledast}_{Y} B$ and $B \rightarrow^{\circledast}_{Y} C$ both hold for $\mathcal{R}$, the reachability formula $A \rightarrow^{\circledast}_{Y} C$ also holds for $\mathcal{R}$.

How can we use the Chain Rule for proving properties of IMPL programs? We can use it to prove properties about *sequential compositions* of the form *stmt stmt'*. How so? Assume that $\vec{x}$ are the program variables read and/or affected by *stmt stmt'*. We first prove, say,

$$< \mathit{stmt} \rightsquigarrow K' \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS > \varphi_1 \rightarrow^{\circledast} < K' \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS > \varphi_2$$

parametric on K' and $\vec{Y}$ and then prove

$$< \mathit{stmt}' \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X'} * VS > \varphi_2 \rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X''} * VS > \varphi_3$$

parametric on K and $\vec{Y}$. Then, thanks to the ChainRule, we have proved:

$$< \mathit{stmt}\ \mathit{stmt}' \rightsquigarrow K \mid TS \;\&\; \vec{x} \mapsto \vec{X} * VS > \varphi_1 \rightarrow^{\circledast} < K \mid TS \;\&\; \vec{x} \mapsto \vec{X''} * VS > \varphi_3$$

parametric on K and $\vec{Y}$. How so? Just by first applying to the first formula the constructor substitution $\{K' \mapsto \mathit{stmt}' \rightsquigarrow K\}$, and then applying the Chain Rule.
Note that this method *is directly based* on the Chain Rule of reachability logic, which is theory-generic: its correctness does not depend at all on the particularities of IMPL.

In summary, this section has presented several *compositional methods* that can be quite useful in arriving at a *proof plan* to prove some desired properties about an IMPL program P. These methods work by *decomposing* a program P into its *subprograms*, and likewise decomposing the original properties for P into simpler properties about its subprograms. The key point is that proving program properties is non-trivial and requires *careful thinking*. The above proof methods can serve as a *guide* to *help* our thinking. *Not* to replace it.

5.2 Loop Invariants

To better understand *loop invariants*, let's focus on the *migrate* program defined in section 4:

$$\mathbf{P} := \mathbf{while}\ (!\mathit{empty}(x))\ \{y =_l y\ \$\colon \mathit{first}(x)\ ;\ x =_l \mathit{rest}(x)\ ;\}$$

Again, we will use the initial store VS_0:

$$VS_0 := x \mapsto X \ * \ y \mapsto Y \ * \ z \mapsto Z$$

And final store VS:

$$VS := x \mapsto X' \ * \ y \mapsto Y' \ * \ z \mapsto Z'$$

The loop invariant is a property that holds before and after the loop executes, and therefore we would like to analyze what changes throughout the loop and what remains constant. We know that the value of z remains unchanged; therefore the logical variable Z should be a parameter of any properties stated about this program. Instead, the values of x and y are updated in each iteration. We would like to capture the way in which the values of these variables change as a static property of the loop, i.e., as *loop invariant*.

While static methods can determine some loop invariants, we don't consider this a strong enough approach to prove interesting properties. Similarly, while tools built upon the semantics of a programming language can be very powerful, they are still limited when compared to a developer with intimate knowledge of the program at hand.

If we focus on the body of our loop, we can in fact see that an element is appended to y in every iteration, whereas x loses an element in each iteration. We therefore know *what* changes during each iteration, and we know *how* it changes; but we need to know the *initial configuration* of the program. This is where we have some creative control. We could assume that y is initialized to the empty list, and that the values of x and z are the *same*: a chosen nonempty list X. We will then define an interesting invariant that hopefully captures the idea we had in mind when writing this program, which sets y to the list x and consumes x in the process. Instead, we choose to generalize the initial configuration according to section 5.1.3 (that is, we consider a precondition that *generalizes* and therefore *subsumes* the one we originally had in mind) and assume that x and y are initialized to arbitrary values X and Y and z to $Y\$X$.

Understanding *what* changes and *how* it changes is the first step, but we need to dive deeper into *how* and understand the implications of the changes and find our precondition. In our case, we begin by analyzing the first iteration of the loop. During this iteration, we take the first element of the list X stored in x, which we will call X_0, and append it to the end of the list stored in y, so that y now contains the value $Y^1 = Y\$X_0$. Following this, x is assigned to the rest of X, meaning that x now has the value $X^1 = X_1\$X_2\$\ldots\$X_n$. After this iteration, we can see that, in fact, $Y\$X = Y\$X_1\$X_2\$\ldots\$X_n = Y^1\$X^1$. If we follow this pattern, we can conclude that, after iteration k, we have $Y^k = Y\$X_1\$\ldots\$X_k$ and $X^k = X_{k+1}\$\ldots\$X_n$, so $Y\$X = Y^k\$X^k$. Since k is arbitrary, we can conclude that this relationship holds at any iteration. It is at this point that we consider z, which is an auxiliary variable we use to represent this relationship.

It can represent it because it was initialized to $Z = Y\$X$ and, since it is not mentioned in the loop, its value does not change at all. But, as pointed out above, we always have $Z = Y^k\$X^k$ for any k. Therefore, our desired *loop invariant*, with data parameter Z, is just the formula $I(Z, X, Y) \equiv (Z = Y\$X)$.

Next, let us consider the midcondition of this loop invariant. As discussed in section 5.1.3, if $I(\vec{Y}, \vec{X})$ is our loop invariant, appearing in the precondition, with data parameters $\vec{Y}$, then the midcondition is $I(\vec{Y}, \vec{X'})$ with $\vec{X'}$ the values of the *program variables* after executing the program. Since in our case the data parameter is Z and the program variables are x and y, our formula for the midcondition is $I(Z, X', Y') \equiv (Z = Y'\$X')$.

5.2.1 Setting Up the Proof of Migrate

To begin the proof, we must first load the specification of IMPL, and then load the reachability logic tool that is the basis of our prover.

```
load impl.maude
load rltool.maude
```

Next, we select the module containing the semantics of IMPL for execution.

```
(select IMPL-SEMANTICS .)
```

We must specify the backend symbolic reasoning tools used by the reachability logic prover for: (i) contextual rewriting, (ii) variant unification, and (iii) variant satisfiability. These are needed by the underlying reachability logic prover and must always be used for proofs in the IMPL prover.

```
(use tool conrew for validity on IMPL-SYNTAX+MUL
                              with FOFORMSIMPLIFY-IMP-IMPL .)
(use tool varunif for varunif on FVP-NAT .)
(use tool varsat  for unsatisfiability on IMPL-SYNTAX .)
```

Now all that is left is to specify the set of terminating states, declare the variables, and add our invariant as a proof goal, as noted in section 4.

```
(def-term-set (< done | St:Store >) | true .)
(declare-vars (X:List) U (X':List) U (Y:List) U (Y':List) U
              (Z:List) U (Z':List) U (K:Continuation) .)
(add-goal migrate :
 (< while (! empty(x)) { y =l y $: first(x) ; x =l rest(x) ; } ~> K
     | (x |-> TList * y |-> TList * z |-> TList)
     & (x |-> X * y |-> Y * z |-> Z) >)
     | (Z) = (Y $ X)
     =>
     (< K | (x |-> TList * y |-> TList * z |-> TList)
     & (x |-> X' * y |-> Y' * z |-> Z') >)
     | (Z) = (Y' $ X') .)
```

Our invariant could have been specified in several ways. We choose invariants that, after several applications by the prover of the **Step** and **Axiom** rules, can easily reach and match the postcondition, so that the proof can be then finished with a last application of the **Subsumption** inference rule. In this way, we can rely on the prover to automatically close the proof, rather than having to use manual intervention to complete the proof.

For example, we use the fresh abstract variables X', Y', and Z', rather than explicitly stating that $x \mapsto nil$, or $y \mapsto X$, or $z \mapsto Z$ in the postcondition. This more general way to specify our goal (where relationships between logical variables are always *moved* to the *data constraint part* of the pattern predicate), allows the tool to have more freedom when performing matches to check needed subsumptions between pattern predicates: for example, when applying the **Axiom** or **Subsume** inference rules of reachability logic. In particular, the invariant $(Z) = (Y \$ X)$ in the precondition, and its variable-renamed version $(Z) = (Y' \$ X')$ in the postcondition are both expressed as *data constraints* in the data constraint part of the respective pattern predicates; we use *different logical variables for different program variables*: X' for x, Y' for y, and Z' for z, so that there is a very simple renaming match between the pre- and postcondition of the loop invariant. The objective of the user should be to generalize and abstract the goals and the way data constraints are expressed in such a way that the prover can use pattern matching to automatically complete most, if not all, of the proof.

5.2.2 Proof of Migrate

At this point we are ready to begin our proof. We do so with the *start-proof* command.

```
(start-proof .)
```

Once the proof is started, the current goal should be displayed as the only currently active goal. As we perform proof steps, we will notice that the list of active proof goals changes. We can utilize the *auto* command to automatically perform these proof steps, most of which either apply the **Step** rule, which amounts to *symbolically executing* one step in the continuation semantics of the program[3] by applying one of the *semantic rules* in $\mathcal{R}^{FG}_{IMPL}$, or attempt to close a proof goal through application of the **Axiom** and/or **Subsumption** rules.

```
(auto .)
```

In the current version of the prover, it takes 35 calls to *auto* before a full iteration of the loop is executed. It is at this point that we can close the proof using the **Axiom** inference rule followed by **Subsumption**. Specifically, since

[3]Recall that here we are using the fine-grain semantics $\mathcal{R}^{FG}_{IMPL}$ of IMPL, so "one step" means either the application of one continuation-transforming equation in E_{IMPL} viewed as a semantic rule, or the application of a semantic rule in R_{IMPL}.

we started with the loop invariant as our only goal, this is a case of a *self axiom* application —i.e., the application of the original goal `migrate` as an *axiom* to one of its children or "descendent" goals— with another call to *auto*.

```
(auto .)
```

To ensure that the proof has been completed, the prover should halt with:

```
Proof Completed.
Action consumed 1 goals and generated 0 goals
```

and should not generate any fatal errors along the way. With this, we have completed the proof. Alternatively, we could use the command

```
(auto* .)
```

to automatically run the proof steps until the prover gets stuck; for this goal, the prover will not get stuck and can automatically completely the proof.

5.2.3 Full Proof Script for Migrate

```
load impl.maude

load rltool.maude

(select IMPL-SEMANTICS  .)
(use tool conrew for validity on IMPL-SYNTAX+MUL
                 with FOFORMSIMPLIFY-IMP-IMPL .)
(use tool varunif for varunif on FVP-NAT .)
(use tool varsat  for unsatisfiability on IMPL-SYNTAX .)
(def-term-set (< done | St:Store >) | true .)
(declare-vars (X:List) U (X':List) U (Y:List) U (Y':List) U
              (Z:List) U (Z':List) U (K:Continuation) .)
(add-goal migrate :
 (< while (! empty(x)) { y =l y $: first(x) ; x =l rest(x) ; } ~> K
     | (x |-> TList * y |-> TList * z |-> TList)
     & (x |-> X * y |-> Y * z |-> Z) >)
     | (Z) = (Y $ X)
     =>
     (< K | (x |-> TList * y |-> TList * z |-> TList)
     & (x |-> X' * y |-> Y' * z |-> Z') >)
     | (Z) = (Y' $ X') .)
(start-proof .)
(auto* .)
```

5.3 List Reverse

To further understand this methodology, we explore the following program *reverse*, which is slightly more complex program to reverse the contents of a list.

In this example, we show how we can add operations to the underlying data types in order to prove more interesting program properties. This is a good illustration of the crucial distinction between *system specification* and *property specification*. For IMPL, the system specification is its continuation formal semantics, specified as the rewrite theory $\mathcal{R}^{FG}_{IMPL}$, whereas property specifications are reachability logic formulas. However, to express some program properties as reachability formulas, the original system specification $\mathcal{R}^{FG}_{IMPL}$, which contains the functional module specifying the data types used in IMPL as the initial algebra of an equational subtheory $(\Sigma_D, E_D \cup B_D)$, while sufficient for simple programs like *migrate*, will in general be insufficient: to even formally *say* what a program is *doing*, e.g., reversing a list, or computing the factorial function, we will need to *extend* the data type theory $(\Sigma_D, E_D \cup B_D)$ with additional function definitions formally specifying the needed functions at the *mathematical level*.

We prove a nontrivial property of *reverse*, which states that this program actually computes what we had in mind, which here means it reverses a list. This program is defined as:

$$\mathbf{P} := \mathbf{while}\ (!empty(x))\ \{y =_l first(x)\ \$\colon y\ ;\ x =_l rest(x)\ ;\}$$

with an initial store VS_0 and final store VS:

$$VS_0 := x \mapsto X\ *\ y \mapsto Y\ *\ z \mapsto Z$$
$$VS := x \mapsto X'\ *\ y \mapsto Y'\ *\ z \mapsto Z$$

We assume that z is a copy of the original list that is not modified.

5.3.1 Finding a Reachability Formula

Finding a reachability formula for this program is a bit trickier. Of course, one could say $empty(X)$ is the postcondition, or find some other uninteresting postcondition, but what is our invariant? Unfortunately, a static look at this program will not yield any interesting preconditions. Fortunately, though, we can use the semantic rules of IMPL and our understanding of the language to determine that, in fact, this program reverses the list x and stores it in y.

With that in mind, we know that our invariant should capture the relationship between z, the original list, x, the original list with some elements removed from the head, and y, the partially reversed list. Since y is a reversed fragment of z, we conclude that the invariant in question is $Z = rev(Y)\ \$\ X$, where rev is the list reverse function at the mathematical level.

5.3.2 Extending the IMPL Specification with Additional Functions

IMPL, and therefore the original language of the IMPL Prover, supports a small, yet rich, set of arithmetic, list and boolean operations. But there is an infinite set of many other *computable functions* that programs in IMPL

can compute in an *imperative fashion*: that is the whole point of IMPL as a Turing-complete language! Of course, the overwhelming majority of all those computable functions are *not* expressible as operations or terms in the equational theory $(\Sigma_D, E_D \cup B_D)$ supporting only the basic operations of IMPL. But, since Maude functional modules are also a Turing-complete language that supports the *mathematical* definition of any computable function on numbers, lists, booleans, or any other user-definable data type, any computable functions on numbers or lists that can be programmed in an imperative fashion in IMPL can be specified in a Maude functional module that *protects* the functional module specified by $(\Sigma_D, E_D \cup B_D)$. For the case of our *reverse* program, we can define a new Maude module that includes both the Maude specification of IMPL's continuation semantics $\mathcal{R}^{FG}_{IMPL}$ *and* the mathematical definition of the *rev* function. We can do this in the following way:

First, we load our IMPL definition (containing a functional submodule for $(\Sigma_D, E_D \cup B_D)$), so that we can define operations over lists.

```
load impl.maude
```

Next, we define a module that protects the theory of IMPL data, which we have called *IMPL-LIST*. We define *rev* as expected, but note that we use the tag *metadata* with value 90. This tag is necessary for the underlying prover; should one add more operations, they must add them with different metadata values, which should be greater than 90.

```
fmod REV is
  pr IMPL-LIST .

  var L : List . var N : Nat .

  op rev : List -> List [metadata "90"] .

  eq rev(nil) = nil .
  eq rev(N $ L) = rev(L) $ N .
endfm
```

Lastly, we create two modules, one syntax and one semantics module, which will be used for contextual rewriting and variant satisfiability as mentioned in section 5.2.1.

```
mod REV+IMPL-SYNTAX+MUL is
  pr REV .
  pr IMPL-SYNTAX+MUL .
endm

mod REV+IMPL-SEMANTICS is
  pr REV .
  pr IMPL-SEMANTICS .
endm
```

5.3.3 Finding the Loop Invariant for Reverse

With our new function, *rev*, we can finally craft and reason about an interesting loop invariant. Again, we must analyze our program and carefully consider *what* changes and *how* it changes.

A quick glance will tell us that, among the three variables x, y, and z, only two list assignment statements occur, corresponding to x and y. What are these assignments doing? Well, x is assigned the rest of its contents in each iteration; therefore its contents will lose its first element in each iteration. We can safely conclude that X' will be a sublist of X when the program terminates. We are not done yet, though, as we note that our loop will finish executing only *if* the contents of x is empty, and with that we can say that, upon termination, $isEmpty(X')$ is *true*.

Turning our attention to y, what can we say about its structure? If we consider a starting configuration where $x \mapsto X$, where $X = X_1\$X_2\$\ldots\$X_n$, and where $y \mapsto Y$, where $Y = nil$, then this becomes quite clear. After the first iteration of the loop, we have $y \mapsto X_1$, and then $y \mapsto X_2\$X_1$ in the next iteration, and so on. Eventually, of course, we will terminate with $Y = rev(X)$.

Now that we know *what* changes, we need to focus on *how*. At some iteration k, we will have $x \mapsto X' \ * \ y \mapsto Y' \ * \ z \mapsto Z$, where $X' = X_{k+1}\$X_{k+2}\$\ldots\$X_n$ and $Y' = X_k\$X_{k-1}\$\ldots\$X_1$, and of course $Z = X_1\$\ldots\$X_n$. How can we relate these three variables? If we look at the original X, we can see that it is split across X' and Y'. Of course, X' is a sublist of X, but Y' is not. Ah, but it *is* a sublist of $rev(X)$, and therefore $rev(Y')$ is a sublist of X, and is clearly disjoint from X'. We can then say that we *always have*: $X = rev(Y')\ \$\ X'$, which is equivalent to stating that $Z = rev(Y')\ \$\ X'$, since we assumed that $Z = X$. Since this identity holds for an arbitrary loop iteration k, we have therefore found our desired loop invariant:

$$\begin{aligned} reverse : <\mathbf{P} \rightsquigarrow K \mid TS \ \&\ VS_0 > \mid Z = rev(Y)\ \$\ X \\ \rightarrow^{\circledast} < K \mid TS \ \&\ VS > \mid Z = rev(Y')\ \$\ X' \end{aligned}$$

5.3.4 Proving of the Invariant for Reverse

The next section should be familiar from the previous example, with minor changes to support our new syntax and semantics modules:

```
load rltool.maude

(select REV+IMP-SEMANTICS  .)
(use tool conrew for validity on REV+IMP-SYNTAX+MUL
                  with FOFORMSIMPLIFY-IMP-IMPL .)
(use tool varunif for varunif          on FVP-NAT .)
(use tool varsat  for unsatisfiability on IMP-SYNTAX .)
```

```
(def-term-set (< done | St:Store >) | true .)
(declare-vars (X:List) U (X':List) U (Y:List) U (Y':List)
                       U (Z:List) U (Z':List) U (K:Continuation) .)
```

All that is left to set up the proof is to add the goal itself. We can use our newly defined *rev* operator to define our reachability formula as expected.

```
(add-goal reverse :
        (< while (! empty(x))
             { y =l first(x) $: y ; x =l rest(x) ; }
             ~> K
        | (x |-> TList * y |-> TList * z |-> TList)
        & (x |-> X * y |-> Y * z |-> Z) >) | (Z) = (rev(Y) $ X) =>
        (< K
        | (x |-> TList * y |-> TList * z |-> TList)
        & (x |-> X' * y |-> Y' * z |-> Z') >)
                                            | (Z) = (rev(Y') $ X') .)
```

We can close this goal and all its subgoals with 35 calls *auto*, and the proof is complete.

5.3.5 Full Proof of Reverse

```
load impl.maude

fmod REV is
  pr IMPL-LIST .

  var L : List . var N : Nat .

  op rev : List -> List [metadata "90"] .

  eq rev(nil) = nil .
  eq rev(N $ L) = rev(L) $ N .
endfm

mod REV+IMPL-SYNTAX+MUL is
  pr REV .
  pr IMPL-SYNTAX+MUL .
endm

mod REV+IMPL-SEMANTICS is
  pr REV .
  pr IMPL-SEMANTICS .
endm

load rltool.maude
```

```
(select REV+IMP-SEMANTICS  .)
(use tool conrew for validity on REV+IMP-SYNTAX+MUL
                  with FOFORMSIMPLIFY-IMP-IMPL .)
(use tool varunif for varunif          on FVP-NAT .)
(use tool varsat  for unsatisfiability on IMP-SYNTAX .)
(def-term-set (< done | St:Store >) | true .)
(declare-vars (X:List) U (X':List) U (Y:List) U (Y':List)
                      U (Z:List) U (Z':List) U (K:Continuation) .)

(add-goal reverse :
        (< while (! empty(x))
             { y =l first(x) $: y ; x =l rest(x) ; }
             ~> K
        | (x |-> TList * y |-> TList * z |-> TList)
        & (x |-> X * y |-> Y * z |-> Z) >) | (Z) = (rev(Y) $ X) =>
        (< K
        | (x |-> TList * y |-> TList * z |-> TList)
        & (x |-> X' * y |-> Y' * z |-> Z') >)
                                          | (Z) = (rev(Y') $ X')  .)

(start-proof .)
(auto* .)
```

6 Conclusions

We have introduced the IMPL as a simple programming language on which to explain, by means of easy to follow examples, a number of important concepts in the specification and deductive verification of properties of imperative programs. These tutorial notes assume a reader already familiar with basic concepts about rewriting logic and reachability logic, but not yet familiar with either the formal semantics of programming languages or the verification of imperative programs. These notes try to provide a tutorial introduction to both of those topics and to help the reader get some initial experience verifying properties of programs. Several possibilities for further reading might be quite natural. For example, the continuation-based style is a good choice for language definitions, but definitely not the only one. Since our prospective reader is already familiar with rewriting logic *and*, after reading this tutorial, with continuation style semantic definitions, the reading of the paper [8], which provides a good overview of the main alternative styles of defining the operational semantics of a programming language, and of how all such styles can be naturally expressed in rewriting logic, would broaden the perspective and provide a fairly comprehensive understanding about operational semantics definitions. It would also be worthwhile to read some papers that can help relate the reachability logic approach in more detail with Hoare logic. One could begin with Hoare's seminal paper [1] and

then compare the proof methods of Hoare logic, as described, e.g., in [12], with those presented here. A short word of warning may be helpful here: in an almost universal way, the Hoare logic notation is based on a *systematic confusion between program variables and logical variables*, i.e., between, a program variable x and the logical variable X for the value stored in x. This hangs as an albatross around the neck of Hoare logic formalisms. To appreciate the language-generic power of reachability logic, as opposed to the language-specific nature of Hoare logic, it would also be worthwhile to read some of the papers by researchers in the K framework relating Hoare logic to reachability logic (and the closely related matching logic), as well as showing the breadth of programming languages to which this language-generic approach to program proving has already been applied, e.g., [7, 11]. Finally, for an overview of the Rewriting Logic Semantics Project, the papers [4, 8, 5] can be consulted.

References

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[2] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[3] J. Meseguer. Twenty years of rewriting logic. *J. Algebraic and Logic Programming*, 81:721–781, 2012.

[4] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373:213–237, 2007.

[5] J. Meseguer and G. Rosu. The rewriting logic semantics project: A progress report. *Inf. Comput.*, 231:38–69, 2013.

[6] G. Rosu and T. Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[7] G. Rosu and A. Stefanescu. From Hoare logic to matching logic reachability. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.

[8] T. Serbanuta, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

[9] S. Skeirik, A. Stefanescu, and J. Meseguer. A constructor-based reachability logic for rewrite theories. *Fundam. Inform.*, 173(4):315–382, 2020.

[10] A. Stefanescu, Ştefan Ciobâcă, R. Mereuta, B. M. Moore, T. Serbanuta, and G. Rosu. All-path reachability logic. In *Proc. RTA-TLCA 2014*, volume 8560, pages 425–440. Springer LNCS, 2014.

[11] A. Stefanescu, D. Park, S. Yuwen, Y. Li, and G. Rosu. Semantics-based program verifiers for all languages. In *Proc. OOPSLA 2016*, pages 74–91. ACM, 2016.

[12] G. Winskel. *The Formal Semantics of Programming Languages — An Introduction*. MIT Press, 1994.

A IMPL Continuation Semantics

The full continuation semantics of IMPL is defined below using the following typed variables:

X	:	Id
AE, AE_1, AE_2	:	$AExp$
BE, BE_1, BE_2	:	$BExp$
LE, LE_1, LE_2	:	$LExp$
N, N'	:	Nat
L, L'	:	$List$
S, S'	:	$Stmt$
K	:	$Cont$

A.1 Equations Transforming Programs Into Continuations

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

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

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

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

$$!BE \rightsquigarrow K\ =\ BE \rightsquigarrow ! \rightsquigarrow K$$
$$BE_1\ \textbf{and}\ BE_2 \rightsquigarrow K\ =\ BE_1 \rightsquigarrow \textbf{and}(BE_2) \rightsquigarrow K$$

$$LE_1\ \textbf{\$:}\ LE_2 \rightsquigarrow K\ =\ (LE_1, LE_2) \rightsquigarrow \textbf{\$:} \rightsquigarrow K$$
$$\textbf{first}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{first} \rightsquigarrow K$$
$$\textbf{rest}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{rest} \rightsquigarrow K$$
$$\textbf{last}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{last} \rightsquigarrow K$$
$$\textbf{prior}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{prior} \rightsquigarrow K$$
$$\textbf{empty}(LE) \rightsquigarrow K\ =\ LE \rightsquigarrow \textbf{empty} \rightsquigarrow K$$

A.2 Tuple Continuation Equations

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

$$
\begin{aligned}
&(LE_1, LE_2) \rightsquigarrow K \;=\; LE_1 \rightsquigarrow (\#, LE_2) \rightsquigarrow K \\
&L_1 \rightsquigarrow (\#, LE_2) \rightsquigarrow K \;=\; LE_2 \rightsquigarrow (L_1, \#) \rightsquigarrow K \\
&L_2 \rightsquigarrow (L_1, \#) \rightsquigarrow K \;=\; (L_1, L_2) \rightsquigarrow K
\end{aligned}
$$

A.3 Variable Update and Variable Lookup Semantic Rules

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

$$
\begin{aligned}
&< X \rightsquigarrow K && \mid (TSt * (X \mapsto TNat)) \,\&\, (VSt * (X \mapsto N)) > \\
\rightarrow\ &< N \rightsquigarrow K && \mid (TSt * (X \mapsto TNat)) \,\&\, (VSt * (X \mapsto N)) > \\
&< X \rightsquigarrow K && \mid (TSt * (X \mapsto TList)) \,\&\, (VSt * (X \mapsto L)) > \\
\rightarrow\ &< L \rightsquigarrow K && \mid (TSt * (X \mapsto TList)) \,\&\, (VSt * (X \mapsto L)) >
\end{aligned}
$$

A.4 Arithmetic, Boolean, and List Data Type Rules

$$< (I_1, I_2) \rightsquigarrow +: \rightsquigarrow K \mid St > \rightarrow < I_1 + I_2 \rightsquigarrow K \mid St >$$
$$< (I_1, I_2) \rightsquigarrow *: \rightsquigarrow K \mid St > \rightarrow < I_1 * I_2 \rightsquigarrow K \mid St >$$
$$< (I_1, I_2) \rightsquigarrow -: \rightsquigarrow K \mid St > \rightarrow < I_1 - I_2 \rightsquigarrow K \mid St >$$
$$< (I_1, I_2) \rightsquigarrow <: \rightsquigarrow K \mid St > \rightarrow < I_1 < I_2 \rightsquigarrow K \mid St >$$

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

$$< (L_1, L_2) \rightsquigarrow \$: \rightsquigarrow K \mid St > \rightarrow < L_1 \ \$ \ L_2 \rightsquigarrow K \mid St >$$
$$< L \rightsquigarrow \textbf{first} \rightsquigarrow K \mid St > \rightarrow < \textbf{head}(L) \rightsquigarrow K \mid St >$$
$$< L \rightsquigarrow \textbf{rest} \rightsquigarrow K \mid St > \rightarrow < \textbf{tail}(L) \rightsquigarrow K \mid St >$$
$$< L \rightsquigarrow \textbf{last} \rightsquigarrow K \mid St > \rightarrow < \textbf{final}(L) \rightsquigarrow K \mid St >$$
$$< L \rightsquigarrow \textbf{prior} \rightsquigarrow K \mid St > \rightarrow < \textbf{prefix}(L) \rightsquigarrow K \mid St >$$
$$< nil \rightsquigarrow \textbf{empty} \rightsquigarrow K \mid St > \rightarrow < true \rightsquigarrow K \mid St >$$
$$< N \ \$ \ L \rightsquigarrow \textbf{empty} \rightsquigarrow K \mid St > \rightarrow < false \rightsquigarrow K \mid St >$$
$$< (L_1, L_2) \rightsquigarrow \textbf{\$:} \rightsquigarrow K \mid St > \rightarrow < L_1 \ \$ \ L_2 \rightsquigarrow K \mid St >$$

A.5 Branching Semantic Rules

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