A Rewriting Logic Approach to Operational Semantics*

Traian Florin Şerbănuţă, Grigore Roşu and José Meseguer

Department of Computer Science,
University of Illinois at Urbana-Champaign.
{tserban2,grosu,meseguer}@cs.uiuc.edu

Abstract

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

1 Introduction

This paper is part of the rewriting logic semantics (RLS) project (see [54, 55] and the references there). The broad goal of the project is to develop a tool-supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on *rewriting logic* [48]. Any logical framework worth its salt should be evaluated in terms of its expressiveness and flexibility. Therefore, a very pertinent question is: how does RLS express other approaches to operational semantics? In particular, how well can it express various approaches in the SOS tradition? The goal of this paper is to provide a careful answer to these questions. Partial answers, giving detailed comparisons with specific approaches have appeared elsewhere. For example, [46] and [88] provide comparisons with standard SOS [68]; [53] compares RLS with both standard SOS and Mosses' modular structural operational semantics (MSOS) [63]; and [48] compares RLS with chemical abstract machine (Cham) semantics [7]. However, no comprehensive comparison encompassing most approaches in the SOS tradition has been given to date. To make our ideas more concrete, in this paper we use a simple programming language, show how it is expressed in each different definitional style, and how that style is captured as a rewrite theory in the RLS framework. We furthermore prove correctness theorems showing the faithfulness of the RLS representation for each style. Even though we exemplify the ideas with a simple language for concreteness' sake, the process of representing each definitional style in RLS is completely general and automatable, and in some cases like MSOS has already been autometed [18]. The range of styles covered includes: big-step (or natural) SOS semantics; small-step SOS semantics; MSOS semantics; context-sensitive reduction semantics; continuation-based semantics; and Cham semantics.

*Supported in part by NSF grants CCF-0234524, CCF-0448501, CNS-0509321, and CNS-05-24516; and by ONR Grant N00014-02-1-0715.

1.1 Challenges

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

1. **Q**: *Can you handle standard SOS?*

   A: As illustrated in Sections 5 and 6 for our example language, and shown more generally in [46, 88, 53] using somewhat different representations, both big-step and small-step SOS definitions can be expressed as rewrite theories in RLS. Furthermore, as illustrated in Section 7 for our language, and systematically explained in [53], MSOS definitions can also be faithfully captured in RLS.

2. **Q**: *Can you handle context-sensitive reduction?*

   A: There are two different questions implicit in the above question: (i) how are approaches to reduction semantics based on evaluation contexts (e.g., [93]) represented as rewrite theories? and (ii) how does RLS support context-sensitive rewriting in general? We answer subquestion (i) in Section 8, where we illustrate with our example language a general method to handle evaluation contexts in RLS. Regarding subquestion (ii), it is worth pointing out that, unlike standard SOS, because of its congruence rule, rewriting logic *is* context-sensitive and, furthermore, using *frozen* operator arguments, reduction can be blocked on selected arguments (see Section 2).

3. **Q**: *Can you handle higher-order syntax?*

   A: It is well-known that higher-order syntax admits first-order representations, such as explicit substitution calculi and de Bruijn numbers, e.g., [1, 6, 79]. However, the granularity of computations is changed in these representations; for example, a single β-reduction step now requires additional rewrites to perform substitutions. In rewriting logic, because computation steps happen in equivalence classes modulo equations, the granularity of computation remains the same, because all explicit substitution steps are equational. Furthermore, using explicit substitution calculi such as CINNI [79], all this can be done automatically, keeping the original higher-order syntax not only for λ-abstraction, but also for any other name-binding operators.

4. **Q**: *What about continuations?*

   A: Continuations [32, 69] are traditionally understood as higher-order functions. Using the above-mentioned explicit calculi they can be represented in a first-order way. In Section 9 we present an alternative view of continuations that is intrinsically first-order, and prove a theorem showing that, for our language, first-order continuation semantics and context-sensitive reduction semantics are equivalent as rewrite theories in RLS.

5. **Q**: *Can you handle concurrency?*

   A: One of the strongest points of rewriting logic is precisely that it is a logical framework for concurrency that can naturally express many different concurrency models and calculi [49, 47]. Unlike standard SOS, which forces an interleaving semantics, true concurrency is directly supported. We illustrate this in Section 10, where we explain how Cham semantics is a particular style within RLS.

6. **Q**: *How expressive is the framework?*

   A: RLS is truly a framework, which does not force on the user any particular definitional style. This is illustrated in this paper by showing how quite different definitional styles can be faithfully captured in RLS. Furthermore, as already mentioned, RLS can express a wide range of concurrent languages and calculi very naturally, without artificial encodings. Finally, real-time and probabilistic systems can likewise be naturally expressed [65, 2, 52].

7. **Q**: *Is anything lost in translation?*

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

8. **Q**: *Is the framework purely operational?*

   A: Although RLS definitions are executable in a variety of systems supporting rewriting, rewriting logic itself is a complete logic with both a computational proof theory and a model-theoretic semantics. In particular, any rewrite theory has an *initial model*, which provides inductive reasoning principles to prove properties. What this means for RLS representations of programming languages is that they have *both* an operational rewriting semantics, and a mathematical model-theoretic semantics. For sequential languages, this model-theoretic semantics is an initial-algebra semantics. For concurrent languages, it is a truly concurrent initial-model semantics. In particular, this initial model has an associated Kripke structure in which temporal logic properties can be both interpreted and model-checked [51].

9. **Q**: *What about performance?*

   A: RLS as such is a *mathematical framework*, not bound to any particular rewrite engine implementation. However, because of the existence of a range of high-performance systems supporting rewriting, RLS semantic definitions can *directly* be used as interpreters when executed in such systems. Performance will then depend on both the system chosen and the particular definitional style used. The RLS theory might need to be slightly adapted to fit the constraints of some of the systems. In Section 11 we present experimental performance results for the execution of our example language using various systems for the different styles considered. Generally speaking, this performance figures are very encouraging and show that good performance interpreters can be directly obtained from RLS semantic definitions.

1.2 Benefits

Our skeptical language designer could still say,

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

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

Besides the benefits already mentioned in our answers to questions in Section 1.1, one obvious benefit is that, since rewriting logic is a *computational* logic, and there are state-of-the-art system implementations supporting it, there is *no gap* between RLS operational semantics definition and an implementation. This is an obvious advantage over the typical situation in which one gives a semantics to a language on paper following one or more operational semantics styles, and then, to "execute" it, one implements an interpreter for the desired language following "in principle" its operational semantics, but using one's favorite programming language and specific tricks and optimizations for the implementation. This creates a nontrivial gap between the formal operational semantics of the language and its implementation.

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

A broader, third benefit, of which the above two are special cases, is the availability of *generic tools* for: (i) syntax; (ii) execution; and (iii) formal analysis. The advantages of generic execution tools have been emphasized above. Regarding (i), languages such as Maude [22] support user-definable syntax for RLS theories, which for language design has two benefits. First, it gives a prototype parser for the defined language essentially for free; and second, the language designer can use directly the concrete syntax of the desired language features, instead of the more common, but harder to read, abstract syntax tree (AST) representation. Regarding (iii), there is a wealth of theorem proving and model checking tools for rewriting/equational-based specifications, which can be used directly to prove properties about language definitions. The fact that these formal analysis tools are generic, should not fool one into thinking that they *must* be inefficient. For example, the LTL model checkers obtained for free in Maude from the RLS definitions of Java and the JVM compare favorably in performance with state-of-the-art Java model checkers [29, 31].

A fourth benefit comes from the availability in RLS of what we call the “abstraction dial,” which can be used to reach a good balance between abstraction and computational observability in semantic definitions. The point is which *computational granularity* is appropriate. A small-step semantics opts for very fine-grained computations. But this is not necessarily the only or the best option for all purposes. The fact that an RLS theory’s axioms include both equations and rewrite rules provides the useful “abstraction dial,” because rewriting takes place *modulo* the equations. That is, computations performed by equations are abstracted out and become *invisible*. This has many advantages, as explained in [54]. For example, for formal analysis it can provide a huge reduction in search space for model checking purposes, which is one of the reasons why the Java model checkers described in [29, 31] perform so well. For language definition purposes, this again has many advantages. For example, in Sections 6 and 5, we use equations to define the semantic infrastructure (stores, etc.) of SOS definitions; in Section 8 equations are also used to hide the extraction and application of evaluation contexts, which are “meta-level” operations, carrying no computational meaning; in Section 9, equations are also used to decompose the evaluation tasks into their corresponding subtasks; finally, in Sections 7 and 10, equations of associativity and commutativity are used to achieve, respectively, modularity of language definitions, and true concurrency in chemical-soup-like computations. The point in all these cases is always the same: to achieve the right granularity of computations.

1.3 Outline of the paper

The remaining of this paper is organized as follows. Section 2 presents basic concepts of rewriting logic and recalls its deduction rules and its relationship with equational logic and term rewriting. Section 3 introduces a simple imperative language that will be used in the rest of the paper to discuss the various definitional styles and their RLS representations. Section 4 gather some useful facts about the algebraic representation of stores. Section 5 addresses the first operational semantics style that we consider in this paper, the big-step semantics. Section 6 discusses the small-step SOS, followed by Section 7 which discusses modular SOS. Sections 8 and 9 show how reduction semantics with evaluation contexts and continuation-based semantics can respectively be faithfully captured as RLS theories, as well as results discussing the relationships between these two interesting semantics. Section 10 presents the Cham semantics. Section 11 shows that the RLS theories corresponding to the various definitional styles provide relatively efficient interpreters to the defined languages when executed on systems that provide support for term rewriting. Finally, Section 12 discusses some related work and Section 13 concludes the paper.

2 Rewriting Logic

Rewriting logic [48] is a computational logic that can be efficiently implemented and that has good properties as a general and flexible *logical and semantic framework*, in which a wide range of logics and models of computation can be faithfully represented [46]. In particular, for programming language semantics it provides the RLS framework, of which we emphasize the operational semantics aspects in this paper (for the mathematical aspects of RLS see [55, 54]).

Two key points to explain are: (i) how rewriting logic combines equational logic and traditional term rewriting; and (ii) what the intuitive meaning of a rewrite theory is all about. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with Σ a signature of function symbols, E a set of (possibly conditional) Σ-equations, and R a set of Σ-rewrite rules which in general may be conditional, with conditions involving both equations and rewrites. That is, a rule in R can have the general form

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

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

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

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

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

2.1 Rewriting Logic Deduction

The inference rules below assume a typed setting, in which (Σ, E) is a *membership equational theory* [50] having sorts (denoted s, s', s'', etc.), subsort inclusions, and kinds (denoted k, k', k'', etc.), which gather together connected components of sorts. Kinds allow error terms like 3/0, which has a kind but no sort. Similar inference rules can be given for untyped or simply typed (many-sorted) versions of the logic. Given $\mathcal{R} = (\Sigma, E, R)$, the sentences that $\mathcal{R}$ proves are universally quantified rewrites of the form $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_\Sigma(X)_k$, for some kind k, which are obtained by finite application of the following *rules of deduction*:

- **Reflexivity**. For each $t \in T_\Sigma(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$

- **Equality**. $\dfrac{(\forall X)\ u \longrightarrow v \quad E \vdash (\forall X) u = u' \quad E \vdash (\forall X) v = v'}{(\forall X)\ u' \longrightarrow v'}$

- **Congruence**. For each $f : s_1 \ldots s_n \longrightarrow s$ in Σ, with $t_i \in T_\Sigma(X)_{s_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{s_{j_l}}$, $1 \leq l \leq m$,
$$\frac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad \ldots \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, \ldots, t_{j_1}, \ldots, t_{j_m}, \ldots, t_n) \longrightarrow f(t_1, \ldots, t'_{j_1}, \ldots, t'_{j_m}, \ldots, t_n)}$$

- **Replacement**. For each $\theta : X \longrightarrow T_\Sigma(Y)$ and for each rule in R of the form
$$(\forall X)\ t \longrightarrow t' \ \text{ if } \ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j),$$

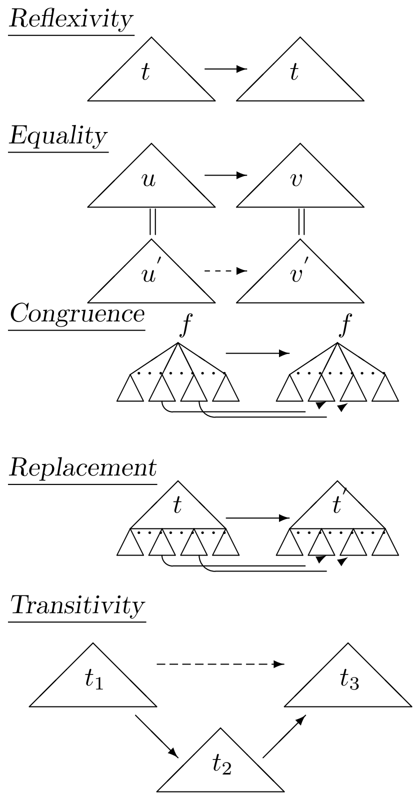

Figure 1: Visual representation of rewriting logic deduction.

$$\frac{(\bigwedge_x (\forall Y)\ \theta(x) \longrightarrow \theta'(x)) \wedge (\bigwedge_i (\forall Y)\ \theta(u_i) = \theta(u_i')) \wedge (\bigwedge_j (\forall Y)\ \theta(w_j) \longrightarrow \theta(w_j'))}{(\forall Y)\ \theta(t) \longrightarrow \theta'(t')}$$

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

- **Transitivity**

$$\frac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$$

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

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

A somewhat more general version of rewriting logic [14] allows rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$, where the additional component ϕ is a function assigning to each function symbol $f \in \Sigma$ with n arguments a subset $\phi(f) \subseteq \{1, \ldots, n\}$ of those argument positions that are *frozen*, that is, positions under which rewriting is forbidden. The above inference rules can then be slightly generalized. Specifically, the **Congruence** rule is restricted to nonfrozen positions $\{j_1, \ldots, j_m\}$, and the substitution θ' in the **Replacement** rule should only differ from θ for variables x in non-frozen positions. The generalized from $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$, makes possible a more expressive control of the possibility of rewriting under contexts already supported by the **Congruence** rule; that is, it endows rewrite theories with flexible context-sensitive rewriting capabilities.

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

The whole point of RLS is then to define the semantics of a programming language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$. RLS uses the fact that rewriting logic deduction is performed *modulo* the equations in $\mathcal{R}_{\mathcal{L}}$ to faithfully capture the desired granularity of a language's computations. This is achieved by making rewriting rules all intended computational steps, while using equations for convenient equivalent structural transformations of the state, or auxiliary "infrastructure" computations, which should not be regarded as computation steps. Note that this does not preclude performing also equational simplification with equations. That is, the set E of equations in a rewrite theory can often be fruitfully decomposed as a disjoint union $E = E_0 \cup A$, where A is a set of *structural axioms*, such as associativity, commutativity and identity of some function symbols, and E_0 is a set of equations that are confluent and terminating *modulo* the axioms A. A rewrite engine supporting rewriting modulo A will then execute both the equations E_0 and the rules R modulo A by rewriting. Under a condition called *coherence* [89], this form of execution then provides a complete inference system for the given rewrite theory (Σ, E, R). However, both conceptually and operationally, the execution of rules R and equations E_0 must be separated. Conceptually, what we are rewriting are E-equivalence classes, so that the E_0-steps become *invisible.* Operationally, the execution of rules R and equations E_0 must be kept separate for soundness reasons. This is particularly apparent in the case of executing *conditional* equations and rules: for a conditional equation it would be *unsound* to use rules in R to evaluate its condition; and for a conditional rule it would likewise be unsound to use rules in R to evaluate the *equational* part of its condition.

There are many systems that either specifically implement term rewriting efficiently, so-called as *rewrite engines*, or support term rewriting as part of a more complex functionality. Any of these systems can be used as an underlying platform for execution and analysis of programming languages defined using the techniques proposed in this paper. Without attempting to be exhaustive, we here only mention (alphabetically) some engines that we are more familiar with, noting that many functional languages and theorem provers provide support for term rewriting as well: ASF+SDF [84], CafeOBJ [26], Elan [8], Maude [21], OBJ [35], and Stratego [90]. Some of these engines can achieve remarkable speeds on today's machines, in the order of tens of millions of rewrite steps per second.

3 A Simple Imperative Language

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

$AExp$	$::=$	$Var \mid \#\ Int \mid AExp$ `+` $AExp \mid AExp$ `-` $AExp \mid AExp$ `*` $AExp \mid$
		$AExp$ `/` $AExp \mid$ `++` Var
$BExp$	$::=$	$\#\ Bool \mid AExp$ `<=` $AExp \mid AExp$ `>=` $AExp \mid AExp$ `==` $AExp \mid$
		$BExp$ `and` $BExp \mid BExp$ `or` $BExp \mid$ `not` $BExp$
$Stmt$	$::=$	`skip` $\mid Var$ `:=` $AExp \mid Stmt$ `;` $Stmt \mid$ `{` $Stmt$ `}` $\mid$
		`if` $BExp$ `then` $Stmt$ `else` $Stmt \mid$ `while` $BExp\ Stmt \mid$ `halt` $AExp$
Pgm	$::=$	$Stmt$ `.` $AExp$

The semantics of `++`x is that of incrementing the value of x in the store and then returning the new value. The increment is done at the moment of evaluation, not after the end of the statement as in C/C++. Also, we assume short-circuit semantics for boolean operations.

This BNF syntax is entirely equivalent to an algebraic order-sorted signature having one (mixfix) operation definition per production, terminals giving the name of the operation and non-terminals the arity. For example, the production defining if-then-else can be seen as an algebraic operation

$$\texttt{if\_then\_else\_} : BExp \times Stmt \times Stmt \to Stmt$$

We will use the following conventions for variables throughout the remaining of the paper: $X \in Var$, $A \in AExp$, $B \in BExp$, $St \in Stmt$, $P \in Pgm$, $I \in Int$, $T \in Bool = \{true, false\}$, $S \in Store$, any of them primed or indexed.

The next sections will use this simple language and will present definitions in various operational semantics styles (big step, small step SOS, MSOS, reduction using evaluation contexts, continuation-based, and Cham), as well the corresponding RLS representation of each definition. We will also characterize the relation between the RLS representations and their corresponding definitional style counterparts, pointing out some strengths and weaknesses for each style. The reader is referred to [42, 68, 63, 93, 7] for further details on the described operational semantics styles.

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

4 Store

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

To define the store, we assume a pairing "binding" constructor "$\_ \mapsto \_$", associating values to variables[1], and an associative and commutative union operation "$\_\ \_$" with $\emptyset$ as its identity to put together such bindings. The equational definition $\mathcal{E}_{Store}$ of operations $\_[\_]$ to retrieve the value of a variable in the store and $\_[\_ \leftarrow \_]$ to update the value of a variable is given by the following equations, that operate modulo the associativity and commutativity of $\_\ \_$:

[1]In general,. one would have both an *environment*, and a *store*, with variables mapped to locations in the environment, and locations mapped to values in the store. However, for the sake of brevity, and given the simplicity of our example language, we do not use environments and map variables directly to values in the store.

$(S\ X \mapsto I)[X] = I$
$(S\ X \mapsto I)[X'] = S[X']$ **if** $X \neq X'$
$(S\ X \mapsto I)[X \leftarrow I'] = S\ X \mapsto I'$
$(S\ X \mapsto I)[X' \leftarrow I'] = S[X' \leftarrow I']\ X \mapsto I$ **if** $X \neq X'$
$\emptyset[X \leftarrow I] = X \mapsto I$

Since these definitions are equational, from a rewriting logic semantic point of view they are invisible: transitions are performed *modulo* these equations. This way we can maintain a coarser computation granularity, while making use of auxiliary functions defined using equations.

A store s is *well-formed* if $\mathcal{E}_{Store} \vdash s = x_1 \mapsto i_1 \ldots x_n \mapsto i_n$ for some $x_j \in \mathit{Var}$ and $i_j \in \mathit{Int}$, for all $1 \leq j \leq n$, such that $x_i \neq x_j$ for any $i \neq j$. We say that a store s is equivalent to a finite partial function $\sigma : \mathit{Var} \overset{\circ}{\rightarrow} \mathit{Int}$, written $s \simeq \sigma$, if s is well-formed and behaves as σ, that is, if for any $x \in \mathit{Var}, i \in \mathit{Int}$, $\sigma(x) = i$ iff $\mathcal{E}_{Store} \vdash s[x] = i$. We recall that, given a store-function σ, $\sigma[i/x]$ is defined as the function mapping x to i and other variables y to $\sigma(y)$.

Proposition 1 *Let $x, x' \in \mathit{Var}$, $i, i' \in \mathit{Int}$, $s, s' \in \mathit{Store}$ and finite partial functions $\sigma, \sigma' : \mathit{Var} \overset{\circ}{\rightarrow} \mathit{Int}$.*

1. *$\emptyset \simeq \perp$ where $\perp$ is the function undefined everywhere.*
2. *$(s\ x \mapsto i) \simeq \sigma$ implies that $s \simeq \sigma[\perp/x]$.*
3. *If $s \simeq \sigma$ then also $s[x \leftarrow i] \simeq \sigma[i/x]$.*

Proof.

1. Trivial, since $\mathcal{E}_{Store} \not\vdash \emptyset[x] = i$ for any $x \in \mathit{Var}, i \in \mathit{Int}$.
2. Consider an arbitrary x'. If $x' = x$, then $\mathcal{E}_{Store} \not\vdash s[x'] = i'$ for any i', since otherwise we would have $\mathcal{E}_{Store} \vdash s = s'\ x \mapsto i'$ which contradicts the well-definedness of $s\ x \mapsto i$. If $x' \neq x$, then $\mathcal{E}_{Store} \vdash s[x'] = (s\ x \mapsto i)[x']$.
3. Suppose $s \simeq \sigma$. We distinguish two cases - if σ is defined on x or if it is not. If it is, then let us say that $\sigma(x) = i'$; in that case we must have that $\mathcal{E}_{Store} \vdash s[x] = i'$ which can only happen if $\mathcal{E}_{Store} \vdash s = s'\ x \mapsto i'$, whence $\mathcal{E}_{Store} \vdash s[x \leftarrow i] = s'\ x \mapsto i$. Let x' be an arbitrary variable in *Var*. If $x' = x$ then
$$\mathcal{E}_{Store} \vdash (s[x \leftarrow i])[x'] = (s'\ x \mapsto i)[x'] = i.$$
If $x' \neq x$ then
$$\mathcal{E}_{Store} \vdash (s[x \leftarrow i])[x'] = (s'\ x \mapsto i)[x'] = s'[x'] = (s'\ x \mapsto i')[x'] = s[x'].$$
If σ is not defined for x, it means that $\mathcal{E}_{Store} \not\vdash s[x] = i$ for any i, whence $\mathcal{E}_{Store} \not\vdash s = s'\ x \mapsto i$. If $\mathcal{E}_{Store} \vdash s = \emptyset$ then we are done, since $\mathcal{E}_{Store} \vdash (x \mapsto i)[x'] = i'$ iff $x = x'$ and $i = i'$. If $\mathcal{E}_{Store} \not\vdash s = \emptyset$, it must be that $\mathcal{E}_{Store} \vdash s = x_1 \mapsto i_1 \ldots x_n \mapsto i_n$ with $x_i \neq x$. This leads to $\mathcal{E}_{Store} \vdash s[x \leftarrow i] = \cdots = (x_1 \mapsto i_1 \ldots x_i \mapsto i_i)[x \leftarrow i](x_{i+1} \mapsto i_{i+1} \ldots x_n \mapsto i_n) = \cdots = \emptyset[x \leftarrow i]s = (x \mapsto i)s = s(x \mapsto i)$.

□

$$\frac{\cdot}{\langle \#I,\sigma\rangle \Downarrow \langle I,\sigma\rangle}$$

$$\frac{\cdot}{\langle X,\sigma\rangle \Downarrow \langle \sigma(X),\sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{++}X,\sigma\rangle \Downarrow \langle I,\sigma[I/X]\rangle},\ \texttt{if}\ \ I=\sigma(X)+1$$

$$\frac{\langle A_1,\sigma\rangle \Downarrow \langle I_1,\sigma_1\rangle,\ \langle A_2,\sigma_1\rangle \Downarrow \langle I_2,\sigma_2\rangle}{\langle A_1+A_2,\sigma\rangle \Downarrow \langle I_1 +_{Int} I_2,\sigma_2\rangle}$$

$$\frac{\cdot}{\langle \#T,\sigma\rangle \Downarrow \langle T,\sigma\rangle}$$

$$\frac{\langle A_1,\sigma\rangle \Downarrow \langle I_1,\sigma_1\rangle,\ \langle A_2,\sigma_1\rangle \Downarrow \langle I_2,\sigma_2\rangle}{\langle A_1\texttt{<=}A_2,\sigma\rangle \Downarrow \langle (I_1 \leq_{Int} I_2),\sigma_2\rangle}$$

$$\frac{\langle B_1,\sigma\rangle \Downarrow \langle true,\sigma_1\rangle,\ \langle B_2,\sigma_1\rangle \Downarrow \langle T,\sigma_2\rangle}{\langle B_1\ \texttt{and}\ B_2,\sigma\rangle \Downarrow \langle T,\sigma_2\rangle}$$

$$\frac{\langle B_1,\sigma\rangle \Downarrow \langle false,\sigma_1\rangle}{\langle B_1\ \texttt{and}\ B_2,\sigma\rangle \Downarrow \langle false,\sigma_1\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle T,\sigma'\rangle}{\langle \texttt{not}\ B,\sigma\rangle \Downarrow \langle not\ (T),\sigma'\rangle}$$

$$\frac{\cdot}{\langle skip,\sigma\rangle \Downarrow \langle \sigma\rangle}$$

$$\frac{\langle A,\sigma\rangle \Downarrow \langle I,\sigma'\rangle}{\langle X\texttt{:=}A,\sigma\rangle \Downarrow \langle \sigma'[I/X]\rangle}$$

$$\frac{\langle St_1,\sigma\rangle \Downarrow \langle \sigma''\rangle,\ \langle St_2,\sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle St_1;St_2,\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle St,\sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \{St\},\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle true,\sigma_1\rangle,\ \langle St_1,\sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2,\sigma\rangle \Downarrow \langle \sigma_2\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle false,\sigma_1\rangle,\ \langle St_2,\sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2,S\rangle \Downarrow \langle \sigma_2\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle false,\sigma'\rangle}{\langle \texttt{while}\ B\ St,\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle true,\sigma_1\rangle,\ \langle St,\sigma_1\rangle \Downarrow \langle \sigma_2\rangle,\ \langle \texttt{while}\ B\ St,\sigma_2\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{while}\ B\ St,\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle St,\bot\rangle \Downarrow \langle \sigma\rangle,\ \langle A,\sigma\rangle \Downarrow \langle I,\sigma'\rangle}{\langle St.A\rangle \Downarrow \langle I\rangle}$$

Table 1: The *BigStep* language definition

$\langle X, S\rangle \rightarrow \langle S[X], S\rangle$		
$\langle \# I, S\rangle \rightarrow \langle I, S\rangle$		
$\langle \texttt{++}X, S\rangle \rightarrow \langle I, S[X \leftarrow I]\rangle$	**if**	$I = S[X] + 1$
$\langle A_1 + A_2, S\rangle \rightarrow \langle I_1 +_{Int} I_2, S_2\rangle$	**if**	$\langle A_1, S\rangle \rightarrow \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \rightarrow \langle I_2, S_2\rangle$
$\langle \# T, S\rangle \rightarrow \langle T, S\rangle$		
$\langle A_1 \texttt{<=} A_2, S\rangle \rightarrow \langle (I_1 \leq_{Int} I_2), S_2\rangle$	**if**	$\langle A_1, S\rangle \rightarrow \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \rightarrow \langle I_2, S_2\rangle$
$\langle B_1 \texttt{ and } B_2, S\rangle \rightarrow \langle T, S_2\rangle$	**if**	$\langle B_1, S\rangle \rightarrow \langle \mathit{true}, S_1\rangle \wedge \langle B_2, S_1\rangle \rightarrow \langle T, S_2\rangle$
$\langle B_1 \texttt{ and } B_2, S\rangle \rightarrow \langle \mathit{false}, S_1\rangle$	**if**	$\langle B_1, S\rangle \rightarrow \langle \mathit{false}, S_1\rangle$
$\langle \texttt{not } B, S\rangle \rightarrow \langle \mathit{not}(T), S'\rangle$	**if**	$\langle B, S\rangle \rightarrow \langle T, S'\rangle$
$\langle \mathit{skip}, S\rangle \rightarrow \langle S\rangle$		
$\langle X \texttt{:=} A, S\rangle \rightarrow \langle S'[X \leftarrow I]\rangle$	**if**	$\langle A, S\rangle \rightarrow \langle I, S'\rangle$
$\langle St_1; St_2, S\rangle \rightarrow \langle S'\rangle$	**if**	$\langle St_1, S\rangle \rightarrow \langle S''\rangle \wedge \langle St_2, S''\rangle \rightarrow \langle S'\rangle$
$\langle \{St\}, S\rangle \rightarrow \langle S'\rangle$	**if**	$\langle St, S\rangle \rightarrow \langle S'\rangle$
$\langle \texttt{if } B \texttt{ then } St_1 \texttt{ else } St_2, S\rangle \rightarrow \langle S_2\rangle$	**if**	$\langle B, S\rangle \rightarrow \langle \mathit{true}, S_1\rangle \wedge \langle St_1, S_1\rangle \rightarrow \langle S_2\rangle$
$\langle \texttt{if } B \texttt{ then } St_1 \texttt{ else } St_2, S\rangle \rightarrow \langle S_2\rangle$	**if**	$\langle B, S\rangle \rightarrow \langle \mathit{false}, S_1\rangle \wedge \langle St_2, S_1\rangle \rightarrow \langle S_2\rangle$
$\langle \texttt{while } B\ St, S\rangle \rightarrow \langle S'\rangle$	**if**	$\langle B, S\rangle \rightarrow \langle \mathit{false}, S'\rangle$
$\langle \texttt{while } B\ St, S\rangle \rightarrow \langle S'\rangle$	**if**	$\langle B, S\rangle \rightarrow \langle \mathit{true}, S_1\rangle \wedge \langle St, S_1\rangle \rightarrow \langle S_2\rangle \wedge \langle \texttt{while } B\ St, S_2\rangle \rightarrow \langle S'\rangle$
$\langle St.A\rangle \rightarrow \langle I\rangle$	**if**	$\langle St, \emptyset\rangle \rightarrow \langle S\rangle \wedge \langle A, S\rangle \rightarrow \langle I, S'\rangle$

Table 2: $\mathcal{R}_{BigStep}$ rewriting logic theory

5 Big-Step Operational Semantics

Introduced as natural semantics in [42], also named relational semantics in [58], or evaluation semantics, big-step semantics is "the most denotational" of the operational semantics. One can view big-step definitions as definitions of functions interpreting each language construct in an appropriate domain.

Big step semantics can be easily represented within rewriting logic. For example, consider the big-step rule defining integer division:

$$\frac{\langle A_1, \sigma\rangle \Downarrow \langle I_1, \sigma_1\rangle, \langle A_2, \sigma_1\rangle \Downarrow \langle I_2, \sigma_2\rangle}{\langle A_1/A_2, \sigma\rangle \Downarrow \langle I_1/_{Int} I_2, \sigma_2\rangle}, \text{ if } I_2 \neq 0.$$

This rule can be automatically translated into the rewrite rule:

$$\langle A_1/A_2, S\rangle \rightarrow \langle I_1/_{Int} I_2, S_2\rangle \textbf{ if } \langle A_1, S\rangle \rightarrow \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \rightarrow \langle I_2, S_2\rangle \wedge I_2 \neq 0$$

The complete[2] big-step operational semantics definition for our simple language, except its `halt` statement (which is discussed at the end of this section), say *BigStep*, is presented in Table 1. To give a rewriting logic theory for the big-step semantics, one needs to first define the various configuration constructs, which are assumed by default in *BigStep*, as corresponding operations extending the signature. Then one can define the rewrite theory $\mathcal{R}_{BigStep}$ corresponding to the big-step operational semantics *BigStep* entirely automatically as shown by Table 2.

Due to the one-to-one correspondence between big-step rules in *BigStep* and rewrite rules in $\mathcal{R}_{BigStep}$, it is easy to prove by induction on the length of derivations the following result:

Proposition 2 *For any $p \in Pgm$ and $i \in Int$, the following are equivalent:*

1. *$BigStep \vdash \langle p\rangle \Downarrow \langle i\rangle$*
2. *$\mathcal{R}_{BigStep} \vdash \langle p\rangle \rightarrow^1 \langle i\rangle$*

[2]Yet, we don't present semantics for equivalent constructs, such as $-, *, /$, `or`.

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

1. $\mathit{BigStep} \vdash \langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ iff $\mathcal{R}_{\mathit{BigStep}} \vdash \langle a, s\rangle \rightarrow^1 \langle i, s'\rangle$ and $s' \simeq \sigma'$,
   for any $a \in \mathit{AExp}, i \in \mathit{Int}, \sigma' : \mathit{Var} \overset{\circ}{\rightarrow} \mathit{Int}$ and $s' \in \mathit{Store}$.

2. $\mathit{BigStep} \vdash \langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$ iff $\mathcal{R}_{\mathit{BigStep}} \vdash \langle b, s\rangle \rightarrow^1 \langle t, s'\rangle$ and $s' \simeq \sigma'$,
   for any $b \in \mathit{AExp}, t \in \mathit{Bool}, \sigma' : \mathit{Var} \overset{\circ}{\rightarrow} \mathit{Int}$ and $s' \in \mathit{Store}$.

3. $\mathit{BigStep} \vdash \langle st, \sigma\rangle \Downarrow \langle \sigma'\rangle$ iff $\mathcal{R}_{\mathit{BigStep}} \vdash \langle st, s\rangle \rightarrow^1 \langle s'\rangle$ and $s' \simeq \sigma'$,
   for any $st \in \mathit{Stmt}, \sigma' : \mathit{Var} \overset{\circ}{\rightarrow} \mathit{Int}$ and $s' \in \mathit{Store}$.

4. $\mathit{BigStep} \vdash \langle p\rangle \Downarrow \langle i\rangle$ iff $\mathcal{R}_{\mathit{BigStep}} \vdash \langle p\rangle \rightarrow^1 \langle i\rangle$,
   for any $p \in \mathit{Pgm}$ and $i \in \mathit{Int}$.

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

1. $\mathit{BigStep} \vdash \langle x\texttt{++}, \sigma\rangle \Downarrow \langle i, \sigma[i/x]\rangle$ iff
   $i = \sigma(x) + 1$ iff
   $\mathcal{E}_{\mathit{Store}} \subseteq \mathcal{R}_{\mathit{BigStep}} \vdash i = s[x] + 1$ iff
   $\mathcal{R}_{\mathit{BigStep}} \vdash \langle x\texttt{++}, s\rangle \rightarrow^1 \langle i, s[x \leftarrow i]\rangle$.
   This completes the proof, since $s[x \leftarrow i] \simeq \sigma[i/x]$, by 3 in Proposition 1.

2. $\mathit{BigStep} \vdash \langle b_1 \ \texttt{and} \ b_2, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$ iff
   ($\mathit{BigStep} \vdash \langle b_1, \sigma\rangle \Downarrow \langle \mathit{false}, \sigma'\rangle$ and $t = \mathit{false}$
   or $\mathit{BigStep} \vdash \langle b_1, \sigma\rangle \Downarrow \langle \mathit{true}, \sigma''\rangle$ and $\mathit{BigStep} \vdash \langle b_2, \sigma''\rangle \Downarrow \langle t, \sigma'\rangle$) iff
   ($\mathcal{R}_{\mathit{BigStep}} \vdash \langle b_1, s\rangle \rightarrow^1 \langle \mathit{false}, s'\rangle$, $s' \simeq \sigma'$ and $t = \mathit{false}$
   or $\mathcal{R}_{\mathit{BigStep}} \vdash \langle b_1, s\rangle \rightarrow^1 \langle \mathit{true}, s''\rangle$, $s'' \simeq \sigma''$, $\mathcal{R}_{\mathit{BigStep}} \vdash \langle b_2, s''\rangle \rightarrow^1 \langle t, \sigma'\rangle$ and $s' \simeq \sigma'$) iff
   $\mathcal{R}_{\mathit{BigStep}} \vdash \langle b_1 \ \texttt{and} \ b_2, s\rangle \rightarrow^1 \langle t, s'\rangle$ and $s' \simeq \sigma'$.

3. $\mathit{BigStep} \vdash \langle \texttt{while} \ b \ st, \sigma\rangle \Downarrow \langle \sigma'\rangle$ iff
   ($\mathit{BigStep} \vdash \langle b, \sigma\rangle \Downarrow \langle \mathit{false}, \sigma'\rangle$
   or $\mathit{BigStep} \vdash \langle b, \sigma\rangle \Downarrow \langle \mathit{true}, \sigma_1\rangle$
   and $\mathit{BigStep} \vdash \langle st, \sigma_1\rangle \Downarrow \langle \sigma_2\rangle$
   and $\mathit{BigStep} \vdash \langle \texttt{while} \ b \ st, \sigma_2\rangle \Downarrow \langle \sigma'\rangle$) iff
   ($\mathcal{R}_{\mathit{BigStep}} \vdash \langle b, s\rangle \rightarrow^1 \langle \mathit{false}, s'\rangle$ and $s' \simeq \sigma'$
   or $\mathcal{R}_{\mathit{BigStep}} \vdash \langle b, s\rangle \rightarrow^1 \langle \mathit{true}, s_1\rangle$, $s_1 \simeq \sigma_1$
   and $\mathcal{R}_{\mathit{BigStep}} \vdash \langle st, s_1\rangle \rightarrow^1 \langle s_2\rangle$, $s_2 \simeq \sigma_2$
   and $\mathcal{R}_{\mathit{BigStep}} \vdash \langle \texttt{while} \ b \ st, s_2\rangle \rightarrow^1 \langle s'\rangle$ and $s' \simeq \sigma'$) iff
   $\mathcal{R}_{\mathit{BigStep}} \vdash \langle \texttt{while} \ b \ st, s\rangle \rightarrow^1 \langle s'\rangle$ and $s' \simeq \sigma'$.

4. $\mathit{BigStep} \vdash \langle st.a\rangle \Downarrow \langle i\rangle$ iff
   $\mathit{BigStep} \vdash \langle st, \bot\rangle \Downarrow \langle \sigma\rangle$ and $\mathit{BigStep} \vdash \langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ iff
   $\mathcal{R}_{\mathit{BigStep}} \vdash \langle st, \emptyset\rangle \rightarrow^1 \langle s\rangle$, $s \simeq \sigma$, $\mathcal{R}_{\mathit{BigStep}} \vdash \langle a, s\rangle \rightarrow^1 \langle i, s'\rangle$ and $s' \simeq \sigma'$ iff
   $\mathcal{R}_{\mathit{BigStep}} \vdash \langle st.a\rangle \rightarrow^1 \langle i\rangle$

This completes the proof. □

The only apparent difference between *BigStep* and $\mathcal{R}_{\mathit{BigStep}}$ is the different notational conventions they use. However, as the above theorem shows, there is a one-to-one correspondence also between their corresponding "computations" (or executions, or derivations). Therefore, $\mathcal{R}_{\mathit{BigStep}}$ actually *is* the big-step operational semantics *BigStep*, not an "encoding" of it. Note that, in order to be faithfully equivalent to *BigStep* computationally, $\mathcal{R}_{BigStep}$ lacks the main strength of rewriting logic that makes it an appropriate formalism for concurrency, namely, that rewrite rules can apply under any context and in parallel (here all rules are syntactically constrained so that they can only apply at the top, sequentially).

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

Weaknesses. Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred. It may be hard or impossible to model concurrent features. It is not modular, e.g., to add side effects to expressions, one must redefine the rules to allow expressions to evaluate to pairs (value-store). It is inconvenient (and non-modular) to define complex control statements; consider, for example, adding halt to the above definition – one needs to add a special configuration *halting*(I), and the following rules:

$$\begin{array}{rcl}
\langle \texttt{halt}\ A, S\rangle \rightarrow \mathit{halting}(I) & \textbf{if} & \langle A.S\rangle \rightarrow \langle I, S'\rangle \\
\langle St_1; St_2, S\rangle \rightarrow \mathit{halting}(I) & \textbf{if} & \langle St_1, S\rangle \rightarrow \mathit{halting}(I) \\
\langle \texttt{while}\ B\ St, S\rangle \rightarrow \mathit{halting}(I) & \textbf{if} & \langle B, S\rangle \rightarrow \langle S'\rangle \wedge \langle St, S'\rangle \rightarrow \mathit{halting}(I) \\
\langle St.A, S\rangle \rightarrow \langle I\rangle & \textbf{if} & \langle St, \emptyset\rangle \rightarrow \mathit{halting}(I)
\end{array}$$

6 Small-Step Operational Semantics

Introduced by Plotkin in [68], also called transition semantics or reduction semantics, small-step semantics captures the notion of one computational step.

One inherent technicality involved in capturing small-step operational semantics as a rewrite theory in a one-to-one notational and computational correspondence is that the rewriting relation is by definition transitive, while the small-step relation is *not* transitive (its transitive closure is defined a posteriori). Therefore, we need to devise a mechanism to "inhibit" rewriting logic's transitive and uncontrolled application of rules. An elegant way to achieve this is to view a small step as a modifier of the current configuration. Specifically, we consider "$\cdot$" to be a modifier on the configuration which performs a "small-step" of computation; in other words, we assume an operation $\cdot\_ : \mathit{Config} \rightarrow \mathit{Config}$. Then, a small-step semantic rule, e.g.,

$$\frac{\langle A_1, S\rangle \rightarrow \langle A_1', S'\rangle}{\langle A_1 + A_2, S\rangle \rightarrow \langle A_1' + A_2, S'\rangle}$$

is translated, again automatically, into a rewriting logic rule, e.g.,

$$\cdot\langle A_1 + A_2, S\rangle \rightarrow \langle A_1' + A_2, S'\rangle \ \textbf{if}\ \cdot\langle A_1, S\rangle \rightarrow \langle A_1', S'\rangle$$

A similar technique is proposed in [53], but there two different types of configurations are employed, one standard and the other "tagged" with the modifier. However, allowing "$\cdot$" to be a modifier rather than a part of a configuration gives more flexibility to the specification – for example, one can specify that one wants two steps simply by putting two dots in front of the configuration.

The complete [3] small-step operational semantics definition for our simple language except its `halt` statement (which is discussed at the end of this section), say *SmallStep*, is presented in Table 3. The corresponding small-step rewriting logic theory $\mathcal{R}_{SmallStep}$ is given in Table 4.

As for big-step semantics, the rewriting under context deduction rule for rewriting logic is again inapplicable, since all rules act at the top, on configurations. However, in *SmallStep* it is not the case that all

[3] However, for brevity's sake, we don't present the semantics of equivalent constructs, such as $-, *, /$, `or`.

$$\frac{\cdot}{\langle X, \sigma\rangle \to \langle(\sigma(X)), \sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{++}X, \sigma\rangle \to \langle I, \sigma[I/X]\rangle}, \ \texttt{if}\ I = \sigma(X) + 1$$

$$\frac{\langle A_1, \sigma\rangle \to \langle A_1', \sigma'\rangle}{\langle A_1 + A_2, \sigma\rangle \to \langle A_1' + A_2, \sigma'\rangle} \qquad \frac{\langle A_2, \sigma\rangle \to \langle A_2', \sigma'\rangle}{\langle I_1 + A_2, \sigma\rangle \to \langle I_1 + A_2', \sigma'\rangle}$$

$$\frac{\cdot}{\langle I_1 + I_2, \sigma\rangle \to \langle I_1 +_{Int} I_2, \sigma\rangle}$$

$$\frac{\langle A_1, \sigma\rangle \to \langle A_1', \sigma'\rangle}{\langle A_1\texttt{<=}A_2, \sigma\rangle \to \langle A_1'\texttt{<=}A_2, \sigma'\rangle} \qquad \frac{\langle A_2, \sigma\rangle \to \langle A_2', \sigma'\rangle}{\langle I_1\texttt{<=}A_2, \sigma\rangle \to \langle I_1\texttt{<=}A_2', \sigma'\rangle}$$

$$\frac{\cdot}{\langle I_1\texttt{<=}I_2, \sigma\rangle \to \langle(I_1 \leq_{Int} I_2), \sigma\rangle}$$

$$\frac{\langle B_1, \sigma\rangle \to \langle B_1', \sigma'\rangle}{\langle B_1\ \texttt{and}\ B_2, \sigma\rangle \to \langle B_1'\ \texttt{and}\ B_2, \sigma'\rangle} \qquad \frac{\langle B, \sigma\rangle \to \langle B', \sigma'\rangle}{\langle \texttt{not}\ B, \sigma\rangle \to \langle \texttt{not}\ B', \sigma'\rangle}$$

$$\frac{\cdot}{\langle \texttt{true and}\ B_2, \sigma\rangle \to \langle B_2, \sigma\rangle} \qquad \frac{\cdot}{\langle \texttt{not true}, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{false and}\ B_2, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle} \qquad \frac{\cdot}{\langle \texttt{not false}, \sigma\rangle \to \langle \texttt{true}, \sigma\rangle}$$

$$\frac{\langle A, \sigma\rangle \to \langle A', \sigma'\rangle}{\langle X\texttt{:=}A, \sigma\rangle \to \langle X\texttt{:=}A', \sigma'\rangle} \qquad \frac{\langle St_1, \sigma\rangle \to \langle St_1', \sigma'\rangle}{\langle St_1; St_2, \sigma\rangle \to \langle St_1'; St_2, \sigma'\rangle}$$

$$\frac{\cdot}{\langle X\texttt{:=}I, \sigma\rangle \to \langle \texttt{skip}, \sigma[I/X]\rangle} \qquad \frac{\cdot}{\langle \texttt{skip}; St_2, \sigma\rangle \to \langle St_2, \sigma\rangle}$$

$$\frac{\cdot}{\langle \{St\}, \sigma\rangle \to \langle St, \sigma\rangle}$$

$$\frac{\langle B, \sigma\rangle \to \langle B', \sigma'\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2, \sigma\rangle \to \langle \texttt{if}\ B'\ \texttt{then}\ St_1\ \texttt{else}\ St_2, \sigma'\rangle}$$

$$\frac{\cdot}{\langle \texttt{if true then}\ St_1\ \texttt{else}\ St_2, \sigma\rangle \to \langle St_1, \sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{if false then}\ St_1\ \texttt{else}\ St_2, \sigma\rangle \to \langle St_2, \sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{while}\ B\ St, \sigma\rangle \to \langle \texttt{if}\ B\ \texttt{then}\ (St; \texttt{while}\ B\ St)\ \texttt{else skip}, \sigma\rangle}$$

$$\frac{\langle St, \sigma\rangle \to \langle St', \sigma'\rangle}{\langle St.A, \sigma\rangle \to \langle St'.A, \sigma'\rangle} \qquad \frac{\langle A, \sigma\rangle \to \langle A', \sigma'\rangle}{\langle \texttt{skip}.A, \sigma\rangle \to \langle \texttt{skip}.A', \sigma'\rangle}$$

$$\frac{\langle P, \bot\rangle \to^* \langle \texttt{skip}.I, \sigma\rangle}{\mathit{eval}(P) \to I}$$

Table 3: The *SmallStep* language definition

$\cdot\langle X, S\rangle \to \langle (S[X]), S\rangle$		
$\cdot\langle \mathtt{++}X, S\rangle \to \langle I, S[X \leftarrow I]\rangle$	**if**	$I = S[X] + 1$
$\cdot\langle A_1 + A_2, S\rangle \to \langle A_1' + A_2, S'\rangle$	**if**	$\cdot\langle A_1, S\rangle \to \langle A_1', S'\rangle$
$\cdot\langle I_1 + A_2, S\rangle \to \langle I_1 + A_2', S'\rangle$	**if**	$\cdot\langle A_2, S\rangle \to \langle A_2', S'\rangle$
$\cdot\langle I_1 + I_2, S\rangle \to \langle I_1 +_{Int} I_2, S\rangle$		
$\cdot\langle A_1 <= A_2, S\rangle \to \langle A_1' <= A_2, S'\rangle$	**if**	$\cdot\langle A_1, S\rangle \to \langle A_1', S'\rangle$
$\cdot\langle I_1 <= A_2, S\rangle \to \langle I_1 <= A_2', S'\rangle$	**if**	$\cdot\langle A_2, S\rangle \to \langle A_2', S'\rangle$
$\cdot\langle I_1 <= I_2, S\rangle \to \langle (I_1 \leq_{Int} I_2), S\rangle$		
$\cdot\langle B_1 \ \mathtt{and}\ B_2, S\rangle \to \langle B_1' \ \mathtt{and}\ B_2, S'\rangle$	**if**	$\cdot\langle B_1, S\rangle \to \langle B_1', S'\rangle$
$\cdot\langle \mathtt{true\ and}\ B_2, S\rangle \to \langle B_2, S\rangle$		
$\cdot\langle \mathtt{false\ and}\ B_2, S\rangle \to \langle \mathtt{false}, S\rangle$		
$\cdot\langle \mathtt{not}\ B, S\rangle \to \langle \mathtt{not}\ B', S'\rangle$	**if**	$\cdot\langle B, S\rangle \to \langle B', S'\rangle$
$\cdot\langle \mathtt{not\ true}, S\rangle \to \langle \mathtt{false}, S\rangle$		
$\cdot\langle \mathtt{not\ false}, S\rangle \to \langle \mathtt{true}, S\rangle$		
$\cdot\langle X := A, S\rangle \to \langle X := A', S'\rangle$	**if**	$\cdot\langle A, S\rangle \to \langle A', S'\rangle$
$\cdot\langle X := I, S\rangle \to \langle \mathtt{skip}, S[X \leftarrow I]\rangle$		
$\cdot\langle St_1; St_2, S\rangle \to \langle St_1'; St_2, S'\rangle$	**if**	$\cdot\langle St_1, S\rangle \to \langle St_1', S'\rangle$
$\cdot\langle \mathtt{skip}; St_2, S\rangle \to \langle St_2, S\rangle$		
$\cdot\langle \{St\}, S\rangle \to \langle St, S\rangle$		
$\cdot\langle \mathtt{if}\ B\ \mathtt{then}\ St_1\ \mathtt{else}\ St_2, S\rangle \to \langle \mathtt{if}\ B'\ \mathtt{then}\ St_1\ \mathtt{else}\ St_2, S'\rangle$	**if**	$\cdot\langle B, S\rangle \to \langle B', S'\rangle$
$\cdot\langle \mathtt{if\ true\ then}\ St_1\ \mathtt{else}\ St_2, S\rangle \to \langle St_1, S\rangle$		
$\cdot\langle \mathtt{if\ false\ then}\ St_1\ \mathtt{else}\ St_2, S\rangle \to \langle St_2, S\rangle$		
$\cdot\langle \mathtt{while}\ B\ St, S\rangle \to \langle \mathtt{if}\ B\ \mathtt{then}\ (St; \mathtt{while}\ B\ St)\ \mathtt{else\ skip}, S\rangle$		
$\cdot\langle St.A, S\rangle \to \langle St'.A, S'\rangle$	**if**	$\cdot\langle St, S\rangle \to \langle St', S'\rangle$
$\cdot\langle \mathtt{skip}.A, S\rangle \to \langle \mathtt{skip}.A', S'\rangle$	**if**	$\cdot\langle A, S\rangle \to \langle A', S'\rangle$
$\mathit{eval}(P) = \mathit{smallstep}(\langle P, \emptyset\rangle)$		
$\mathit{smallstep}(\langle P, S\rangle) = \mathit{smallstep}(\cdot\langle P, S\rangle)$		
$\mathit{smallstep}(\cdot\langle \mathtt{skip}.I, S\rangle) \to I$		

Table 4: $\mathcal{R}_{SmallStep}$ rewriting logic theory

right hand sides are normal forms (this actually is the specificity of small-step semantics). The "$\cdot$" operator introduced in $\mathcal{R}_{SmallStep}$ prevents the unrestricted application of transitivity, and can be regarded as a token given to a configuration to allow it to change to the next step. We use transitivity at the end (rules for *smallstep*) to obtain the transitive closure of the small-step relation by specifically giving tokens to the configuration until it reaches a normal form.

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

Proposition 3 *For any $p \in Pgm$, $\sigma, \sigma' : Var \overset{\circ}{\to} Int$ and $s \in Store$ such that $s \simeq \sigma$, the following are equivalent:*

1. *$SmallStep \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$, and*

2. *$\mathcal{R}_{SmallStep} \vdash \cdot\langle p, s\rangle \to^1 \langle p', s'\rangle$ and $s' \simeq \sigma'$.*

Moreover, the following are equivalent for any $p \in Pgm$ and $i \in Int$:

1. *$SmallStep \vdash \langle p, \bot\rangle \to^* \langle \mathtt{skip}.i, \sigma\rangle$ for some $\sigma : Var \overset{\circ}{\to} Int$, and*

2. *$\mathcal{R}_{SmallStep} \vdash eval(p) \to i$.*

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

1. $SmallStep \vdash \langle a, \sigma\rangle \to \langle a', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle a, s\rangle \to^1 \langle a', s'\rangle$ and $s' \simeq \sigma'$,
   for any $a, a' \in AExp$, $\sigma' : Var \overset{\circ}{\to} Int$ and $s' \in Store$.

2. $SmallStep \vdash \langle b, \sigma\rangle \to \langle b', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle b, s\rangle \to^1 \langle b', s'\rangle$ and $s' \simeq \sigma'$,
   for any $b, b' \in BExp$, $\sigma' : Var \overset{\circ}{\to} Int$ and $s' \in Store$.

3. $SmallStep \vdash \langle st, \sigma\rangle \to \langle st', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle st, s\rangle \to^1 \langle st', s'\rangle$ and $s' \simeq \sigma'$,
   for any $st, st' \in Stmt$, $\sigma' : Var \overset{\circ}{\to} Int$ and $s' \in Store$.

4. $SmallStep \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle p, s\rangle \to^1 \langle p', s'\rangle$ and $s' \simeq \sigma'$,
   for any $p, p' \in Pgm$, $\sigma' : Var \overset{\circ}{\to} Int$ and $s' \in Store$.

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

1. $SmallStep \vdash \langle a_1 + a_2, \sigma\rangle \to \langle a_1 + a_2', \sigma'\rangle$ iff
   $a_1 = i$ and $SmallStep \vdash \langle a_2, \sigma\rangle \to \langle a_2', \sigma'\rangle$ iff
   $a_1 = i$, $R_{SmallStep} \vdash \cdot\langle a_2, s\rangle \to^1 \langle a_2', s'\rangle$ and $s' \simeq \sigma'$ iff
   $R_{SmallStep} \vdash \cdot\langle a_1 + a_2, s\rangle \to^1 \langle a_1 + a_2', s'\rangle$ and $s' \simeq \sigma'$.

2. $SmallStep \vdash \langle \mathtt{not\ true}, \sigma\rangle \to \langle \mathtt{false}, \sigma\rangle$ iff
   $R_{SmallStep} \vdash \cdot\langle \mathtt{not\ true}, s\rangle \to^1 \langle \mathtt{false}, s\rangle$.

3. $SmallStep \vdash \langle st_1; st_2, \sigma\rangle \to \langle st_1'; st_2, \sigma'\rangle$ iff
   $SmallStep \vdash \langle st_1, \sigma\rangle \to \langle st_1', \sigma'\rangle$ iff
   $R_{SmallStep} \vdash \cdot\langle st_1, s\rangle \to^1 \langle st_1', s'\rangle$ and $s' \simeq \sigma'$ iff
   $R_{SmallStep} \vdash \cdot\langle st_1; st_2, s\rangle \to^1 \langle st_1' + st_2, s'\rangle$ and $s' \simeq \sigma'$.

4. $SmallStep \vdash \langle st.a, \sigma\rangle \to \langle st.a', \sigma'\rangle$ iff
   $st = \mathtt{skip}$ and $SmallStep \vdash \langle a, \sigma\rangle \to \langle a', \sigma'\rangle$ iff
   $st = \mathtt{skip}$, $R_{SmallStep} \vdash \cdot\langle a, s\rangle \to^1 \langle a', s'\rangle$ and $s' \simeq \sigma'$ iff
   $R_{SmallStep} \vdash \cdot\langle st.a, s\rangle \to \langle st.a', s'\rangle$ and $s' \simeq \sigma'$.

Let us now move to the second equivalence. For this proof let $\rightarrow^n$ be the restriction of $\mathcal{R}_{\mathit{SmallStep}}$ relation $\rightarrow$ to those pairs which can be provable by exactly applying $n-1$ times the **Transitivity** rule if $n > 0$, or **Reflexivity** for $n = 0$. We first prove the following more general result (suppose $p \in \mathit{Pgm}$, $\sigma : \mathit{Var} \overset{\circ}{\rightarrow} \mathit{Int}$ and $s \in \mathit{Store}$ such that $s \simeq \sigma$):

$$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \rightarrow^n \langle p', \sigma'\rangle \text{ iff } \mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) \rightarrow^n \mathit{smallstep}(\cdot\langle p', s'\rangle) \text{ and } s' \simeq \sigma',$$

by induction on n. If $n = 0$ then $\langle p, \sigma\rangle = \langle p', \sigma'\rangle$ and since $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) = \mathit{smallstep}(\cdot\langle p, s\rangle)$ we are done. If $n > 0$, we have that
$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \rightarrow^n \langle p', \sigma'\rangle$ iff
$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \rightarrow \langle p_1, \sigma_1\rangle$ and $\mathit{SmallStep} \vdash \langle p_1, \sigma_1\rangle \rightarrow^{n-1} \langle p', \sigma'\rangle$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \cdot\langle p, s\rangle \rightarrow \langle p_1, s_1\rangle$ and $s_1 \simeq \sigma_1$ (by 1)
  and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p_1, s_1\rangle) \rightarrow^{n-1} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$ (by the induction hypothesis) iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle p, s\rangle) \rightarrow^1 \mathit{smallstep}(\langle p_1, s_1\rangle)$ and $s_1 \simeq \sigma_1$
  and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p_1, s_1\rangle) \rightarrow^{n-1} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle p, s\rangle) \rightarrow^n \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$.
We are done, since $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) = \mathit{smallstep}(\cdot\langle p, s\rangle)$.

Finally, $\mathit{SmallStep} \vdash \langle p, \bot\rangle \rightarrow^* \langle \mathtt{skip}.i, \sigma\rangle$ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, \emptyset\rangle) \rightarrow \mathit{smallstep}(\cdot\langle \mathtt{skip}.i, s\rangle)$, $s \simeq \sigma$; the rest follows from $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{eval}(p) = \mathit{smallstep}(\langle p, \emptyset\rangle)$ and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle \mathtt{skip}.i, s\rangle) = i$.
$\square$

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

Weaknesses. Each small step does the same amount of computation as a big step in finding the next redex. It does not give a "true concurrency" semantics, that is, one has to chose a certain interleaving (no two rules can be applied at the same time), mainly because reduction is forced to occur only at the top. One of the reasons for introducing SOS was that abstract machines need to introduce new syntactic constructs to decompose the abstract syntax tree, while SOS would and should only work by modifying the structure of the program. We argue that this is not entirely accurate: for example, one needs to have the syntax of boolean values if one wants to have boolean expressions, and needs an `if` mechanism in the above definition to evaluate `while`. The fact that these features are common in programming languages does not mean that the languages which don't want to allow them should be despised. It is still hard to deal with control – for example, consider adding halt to this language. One cannot simply do it as for other ordinary statements: instead, one has to add a corner case (additional rule) to each statement, as shown below:

$$\begin{array}{rl} \cdot\langle \mathtt{halt}\ A, S\rangle \rightarrow \langle \mathtt{halt}\ A', S'\rangle & \quad \textbf{if} \quad \cdot\langle A, S\rangle \rightarrow \langle A', S'\rangle \\ \cdot\langle \mathtt{halt}\ I; St, S\rangle \rightarrow \langle \mathtt{halt}\ I, S\rangle & \\ \cdot\langle \mathtt{halt}\ I.A, S\rangle \rightarrow \langle \mathtt{skip}.I, S\rangle & \end{array}$$

If expressions could also halt the program, e.g., if one adds functions, then a new rule would have to be added to specify the corner case for each halt-related arithmetic or boolean construct. Moreover, by propagating the "halt signal" through all the statements and expressions, one fails to capture the intended computation granularity of halt: it should just terminate the execution in *one step*!

7 MSOS Semantics

MSOS semantics was introduced by Mosses in [62, 63] to deal with the non-modularity issues of small-step and big-step semantics. The solution proposed in *MSOS* involves moving the non-syntactic state components to the arrow labels, plus a discipline of only selecting needed attributes from the states.

A transition in *MSOS* is of the form $P \xrightarrow{u} P'$, where P and P' are program expressions and u is a label describing the structure of the state both before and after the transition. If u is missing, then the state is assumed to stay unchanged. Specifically, u is a record containing fields denoting the semantic components. Modularity is achieved by the record comprehension notation "..." which indicates that more fields could follow but that they are not of interest for this transition. Fields of a label can fall in one of the following categories: *read-only*, *read-write* and *write-only*.

Read-only fields are only inspected by the rule, but not modified. For example, when reading the location of a variable in an environment, the environment is not modified.

Read-write fields come in pairs, having the same field name, except that the "write" field name is primed. They are used for transitions modifying existing state fields. For example, a store field σ can be read and written. This is illustrated by one of the rules for assignment:

$$\frac{\text{unobs}\{\sigma = \sigma_0, \ldots\}}{X\texttt{:=}I \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[I/X],\ldots\}} \texttt{skip}}$$

The above rule says that, if before the transition the store was σ_0, after the transition it will become $\sigma_0[I/X]$, updating X by I. The unobs predicate is used to express that the rest of the state does not change.

Write-only fields are used to record things not analyzable during the execution of the program, such as the output or the trace. Their names are always primed and they have a free monoid semantics – everything written on then is actually added at the end. A good example of the usage of write-only fields would be a rule for defining a `print` language construct:

$$\frac{\text{unobs}\{\mathit{out}' = L, \ldots\}}{\texttt{print}(I) \xrightarrow{\{\mathit{out}'=I,\ldots\}} \texttt{skip}}$$

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

The *MSOS* description of the small-step SOS definition in Table 3 is given in Table 5.

Because the part of the state not involved in a certain rule is hidden through the "..." notation, language extensions can be made modularly. Consider, for example, adding `halt` to the definition in Table 5. What needs to be done is to add another read-write field in the record, say *halt?*, along with the possible values *halted(i)*, for any integer i, and *false*, as the default value, along with a construct `stuck` to block the execution of the program. The rules *MSOS* rules for `halt` are then:

$$\frac{A \xrightarrow{S} A'}{\texttt{halt}\ A \xrightarrow{S} \texttt{halt}\ A'}$$

$$\frac{\text{unobs}\{\mathit{halt?} = \mathit{false}, \ldots\}}{\texttt{halt}\ I \xrightarrow{\{\mathit{halt?}=\mathit{false},\mathit{halt?}'=\mathit{halted}(I),\ldots\}} \texttt{stuck}}$$

$$\frac{\text{unobs}\{\mathit{halt?} = \mathit{halted}(I), \ldots\}}{P \xrightarrow{\{\mathit{halt?}=\mathit{halted}(I),\ldots\}} \texttt{skip}.I}$$

To give a faithful representation of *MSOS* definitions in rewriting logic, we here follow the methodology in [53]. Using the fact that labels describe changes from their source state to their destination state, one can move the labels back into the configurations. That is, a transition step $P \xrightarrow{u} P'$ is modeled as a rewrite step $\cdot\langle P, u^{pre}\rangle \to \langle P', u^{post}\rangle$, where u^{pre} and u^{post} are *records* describing the state before and after the transition. Notice again the use of the "$\cdot$" operator to emulate small steps by restricting transitivity. State records can be specified equationally as wrapping (using a constructor "$\{\_\}$") a set of fields built from *fields* as constructors, using an associative and commutative concatenation operation "$\_,\_$". Fields are constructed from state attributes; for example, the store can be embedded into a field by a constructor "$\sigma : \_$".

Records u^{pre} and u^{post} are computed from u in the following way:

$$\frac{\text{unobs}\{\sigma, \ldots\},\ \sigma(X) = I}{X \xrightarrow{\{\sigma,\ldots\}} I}$$

$$\frac{\text{unobs}\{\sigma = \sigma_0, \ldots\},\ I = \sigma_0(X) + 1}{\texttt{++}X \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[I/X],\ldots\}} I}$$

$$\frac{A_1 \xrightarrow{\mathcal{S}} A_1'}{A_1 + A_2 \xrightarrow{\mathcal{S}} A_1' + A_2} \qquad \frac{A_2 \xrightarrow{\mathcal{S}} A_2'}{I_1 + A_2 \xrightarrow{\mathcal{S}} I_1 + A_2'}$$

$$I_1 + I_2 \rightarrow I_1 +_{Int} I_2$$

$$\frac{A_1 \xrightarrow{\mathcal{S}} A_1'}{A_1\texttt{<=}A_2 \xrightarrow{\mathcal{S}} A_1'\texttt{<=}A_2} \qquad \frac{A_2 \xrightarrow{\mathcal{S}} A_2'}{I_1\texttt{<=}A_2 \xrightarrow{\mathcal{S}} I_1\texttt{<=}A_2'}$$

$$I_1\texttt{<=}I_2 \rightarrow I_1 \leq_{Int} I_2$$

$$\frac{B_1 \xrightarrow{\mathcal{S}} B_1'}{B_1\ \texttt{and}\ B_2 \xrightarrow{\mathcal{S}} B_1'\ \texttt{and}\ B_2} \qquad \frac{B \xrightarrow{\mathcal{S}} B'}{\texttt{not}\ B \xrightarrow{\mathcal{S}} \texttt{not}\ B'}$$

$$\texttt{true and}\ B_2 \rightarrow B_2 \qquad \texttt{not true} \rightarrow \texttt{false}$$

$$\texttt{false and}\ B_2 \rightarrow \texttt{false} \qquad \texttt{not false} \rightarrow \texttt{true}$$

$$\frac{A \xrightarrow{\mathcal{S}} A'}{X\texttt{:=}A \xrightarrow{\mathcal{S}} X\texttt{:=}A'} \qquad \frac{St_1 \xrightarrow{\mathcal{S}} St_1'}{St_1; St_2 \xrightarrow{\mathcal{S}} St_1'; St_2}$$

$$\frac{\text{unobs}\{\sigma = \sigma_0, \ldots\}}{X\texttt{:=}I \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[I/X],\ldots\}} \texttt{skip}} \qquad \texttt{skip}; St_2 \rightarrow St_2$$

$$\{St\} \rightarrow St$$

$$\frac{B \xrightarrow{\mathcal{S}} B'}{\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2 \xrightarrow{\mathcal{S}} \texttt{if}\ B'\ \texttt{then}\ St_1\ \texttt{else}\ St_2}$$

$$\texttt{if true then}\ St_1\ \texttt{else}\ St_2 \rightarrow St_1$$

$$\texttt{if false then}\ St_1\ \texttt{else}\ St_2 \rightarrow St_2$$

$$\texttt{while}\ B\ St \rightarrow \texttt{if}\ B\ \texttt{then}\ (St; \texttt{while}\ B\ St)\ \texttt{else skip}$$

$$\frac{St \xrightarrow{\mathcal{S}} St'}{St.A \xrightarrow{\mathcal{S}} St'.A} \qquad \frac{A \xrightarrow{\mathcal{S}} A'}{\texttt{skip}.A \xrightarrow{\mathcal{S}} \texttt{skip}.A'}$$

Table 5: The *MSOS* language definition

- For unobservable transitions, $u^{pre} = u^{post}$;
- *Read-only* fields of u are added to both u^{pre} and u^{post}.
- *Read-write* fields of u are translated by putting the read part in u^{pre} and the (now unprimed) write part in u^{post}. The assignment rule, for example, becomes:

  $$\cdot\langle X\texttt{:=}I, \{\sigma : S_0, W\}\rangle \rightarrow \langle \texttt{skip}, \{\sigma : S_0[X \leftarrow I], W\}\rangle$$

  Notice that the "..." notation gets replaced by a generic field-set variable W.
- *Write-only* fields $i' = v$ of u are translated as follows: $i : L$, with L a fresh new variable, is added to u^{pre}, and $i : Lv$ is added to u^{post}. For example, the `print` rule above becomes:

  $$\cdot\langle \texttt{print}(I), \{out : L, W\}\rangle \rightarrow \langle \texttt{skip}, \{out : LI, W\}\rangle$$
- When dealing with observable transitions, both state records meta-variables and ... operations are represented in u^{pre} by some variables, while in u^{post} by others. For example, the first rule defining addition in Table 5 is translated into:

  $$\cdot\langle A_1 + A_2, R\rangle \rightarrow \langle A_1' + A_2, R'\rangle \quad \textbf{if} \quad \cdot\langle A_1, R\rangle \rightarrow \langle A_1', R'\rangle$$

The key thing to notice here is that modularity is preserved by this translation. What indeed makes *MSOS* definitions modular is the record comprehension mechanism. A similar comprehension mechanism is achieved in rewriting logic by using sets of fields and matching modulo associativity and commutativity. That is, the extensibility provided by the "..." record notation in *MSOS* is here captured by associative and commutative matching on the W variable, which allows new fields to be added.

The relation between *MSOS* and R_{MSOS} definitions assumes that *MSOS* definitions are in a certain *normal form* [53] and is made precise by the following theorem, strongly relating MSOS and modular rewriting semantics.

Theorem 1 *[53] For each normalized MSOS definition, there is a strong bisimulation between its transition system and the transition system associated to its translation in rewriting logic.*

The above presented translation is the basis for the `Maude-MSOS` tool [18], which has been used to define and analyze complex language definitions, such as Concurrent ML [17].

Table 6 presents the rewrite theory corresponding to the *MSOS* definition in Table 5. The only new variable symbols introduced are R,R', standing for records, and W standing for the remaining of a record.

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

Weaknesses. Control is still not explicit in MSOS, making combinations of control-dependent features (e.g., call/cc) harder to specify [63, page 223].

8 Reduction Semantics with Evaluation Contexts

Introduced in [93], also called context reduction, the evaluation contexts style improves over small-step definitional style in two ways:

1. it gives a more compact semantics to context-sensitive reduction, by using parsing to find the next redex rather than small-step rules; and

$\cdot\langle X, \{\sigma : S, W\}\rangle \to \langle I, \{\sigma : S, W\}\rangle$	**if**	$I = S[X]$
$\cdot\langle \mathtt{++}X, \{\sigma : S_0, W\}\rangle \to \langle I, \{S_0[X \leftarrow I], W\}\rangle$	**if**	$I = S_0[X] + 1$
$\cdot\langle A_1 + A_2, R\rangle \to \langle A_1' + A_2, R'\rangle$	**if**	$\cdot\langle A_1, R\rangle \to \langle A_1', R'\rangle$
$\cdot\langle I_1 + A_2, R\rangle \to \langle I_1 + A_2', R'\rangle$	**if**	$\cdot\langle A_2, R\rangle \to \langle A_2', R'\rangle$
$\cdot\langle I_1 + I_2, R\rangle \to \langle I_1 +_{Int} I_2, R\rangle$		
$\cdot\langle A_1 \mathtt{<=} A_2, R\rangle \to \langle A_1' \mathtt{<=} A_2, R'\rangle$	**if**	$\cdot\langle A_1, R\rangle \to \langle A_1', R'\rangle$
$\cdot\langle I_1 \mathtt{<=} A_2, R\rangle \to \langle I_1 \mathtt{<=} A_2', R'\rangle$	**if**	$\cdot\langle A_2, R\rangle \to \langle A_2', R'\rangle$
$\cdot\langle I_1 \mathtt{<=} I_2, R\rangle \to \langle I_1 \leq_{Int} I_2, R\rangle$		
$\cdot\langle B_1 \ \mathtt{and}\ B_2, R\rangle \to \langle B_1' \ \mathtt{and}\ B_2, R'\rangle$	**if**	$\cdot\langle B_1, R\rangle \to \langle B_1', R'\rangle$
$\cdot\langle \mathtt{true\ and}\ B_2, R\rangle \to \langle B_2, R\rangle$		
$\cdot\langle \mathtt{false\ and}\ B_2, R\rangle \to \langle \mathtt{false}, R\rangle$		
$\cdot\langle \mathtt{not}\ B, R\rangle \to \langle \mathtt{not}\ B', R'\rangle$	**if**	$\cdot\langle B, R\rangle \to \langle B', R'\rangle$
$\cdot\langle \mathtt{not\ true}, R\rangle \to \langle \mathtt{false}, R\rangle$		
$\cdot\langle \mathtt{not\ false}, R\rangle \to \langle \mathtt{true}, R\rangle$		
$\cdot\langle X \mathtt{:=} A, R\rangle \to \langle X \mathtt{:=} A', R'\rangle$	**if**	$\cdot\langle A, R\rangle \to \langle A', R'\rangle$
$\cdot\langle X \mathtt{:=} I, \{\sigma : S_0, W\}\rangle \to \langle \mathtt{skip}, \{\sigma : S_0[X \leftarrow I], W\}\rangle$		
$\cdot\langle St_1; St_2, R\rangle \to \langle St_1'; St_2, R'\rangle$	**if**	$\cdot\langle St_1, R\rangle \to \langle St_1', R'\rangle$
$\cdot\langle \mathtt{skip}; St_2, R\rangle \to \langle St_2, R\rangle$		
$\cdot\langle \{St\}, R\rangle \to \langle St, R\rangle$		
$\cdot\langle \mathtt{if}\ B\ \mathtt{then}\ St_1\ \mathtt{else}\ St_2, R\rangle \to \langle \mathtt{if}\ B'\ \mathtt{then}\ St_1\ \mathtt{else}\ St_2, R'\rangle$	**if**	$\cdot\langle B, R\rangle \to \langle B', R'\rangle$
$\cdot\langle \mathtt{if\ true\ then}\ St_1\ \mathtt{else}\ St_2, R\rangle \to \langle St_1, R\rangle$		
$\cdot\langle \mathtt{if\ false\ then}\ St_1\ \mathtt{else}\ St_2, R\rangle \to \langle St_2, R\rangle$		
$\cdot\langle \mathtt{while}\ B\ St, R\rangle \to \langle \mathtt{if}\ B\ \mathtt{then}\ (St; \mathtt{while}\ B\ St)\ \mathtt{else\ skip}, R\rangle$		
$\cdot\langle St.A, R\rangle \to \langle St'.A, R'\rangle$	**if**	$\cdot\langle St, R\rangle \to \langle St', R'\rangle$
$\cdot\langle \mathtt{skip}.A, R\rangle \to \langle \mathtt{skip}.A', R'\rangle$	**if**	$\cdot\langle A, R\rangle \to \langle A', R'\rangle$

Table 6: R_{MSOS} rewriting logic theory

2. it provides the possibility of also modifying the context in which a reduction occurs, making it much easier to deal with control-intensive features. For example, defining halt is done now using only one rule, $C[\mathtt{halt}\ I] \to I$, preserving the desired computational granularity.

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

$$C[E] \to C[E'] \text{ when } E \to E',$$

capturing the fact that reductions are allowed to take place only in appropriate evaluation contexts. Therefore, an important part of a context reduction semantics is the definition of evaluation contexts, which is typically done by means of a context-free grammar. The definition of evaluation contexts for our simple language is found in Table 7 (we let [] denote the "hole").

In this BNF definition of evaluation contexts, S is a store variable. Therefore, a "top level" evaluation context will also contain a store in our simple language definition. There are also context-reduction definitions which operate only on syntax (i.e., no additional state is needed), but instead one needs to employ some substitution mechanism (particularly in definitions of λ-calculus based languages). The rules following the contexts grammar in Table 7 complete the context reduction semantics of our simple language, say *CxtRed*.

By making the evaluation context explicit and changeable, context reduction is, in our view, a significant improvement over small-step SOS. In particular, one can now define control-intensive statements like `halt` *modularly* and at the desired level of computational granularity. Even though the definition in Table 7 gives one the feeling that evaluation contexts and their instantiation come "for free", the application of the "rewrite in context" rule presented above can be expensive in practice. This is because one needs either to parse/search the entire configuration to put it in the form $C[E]$ for some appropriate C satisfying

$$\begin{array}{rcl} C & ::= & [] \\ & | & \langle C, S\rangle \\ & | & \texttt{skip}.C \mid C.A \\ & | & X\,\texttt{:=}C \mid C;St \mid \texttt{if}\,C\,\texttt{then}\,St_1\,\texttt{else}\,St_2 \mid \texttt{halt}\,C \\ & | & I\texttt{<=}C \mid C\texttt{<=}A \mid C\,\texttt{and}\,B \mid \texttt{not}\,C \\ & | & I + C \mid C + A \end{array}$$

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

$I_1 + I_2 \to (I_1 +_{Int} I_2)$
$\langle P, \sigma\rangle[X] \to \langle P, \sigma\rangle[(\sigma(X))]$
$\langle P, \sigma\rangle[\texttt{++}X] \to \langle P, \sigma[I/X]\rangle[I]$ when $I = \sigma(X) + 1$

$I_1\texttt{<=}I_2 \to (I_1 \leq_{Int} I_2)$
$\texttt{true and } B \to B$
$\texttt{false and } B \to \texttt{false}$
$\texttt{not true} \to \texttt{false}$
$\texttt{not false} \to \texttt{true}$

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

$C[\texttt{skip}.I] \to \langle I\rangle$

Table 7: The *CxtRed* language definition

the grammar of evaluation contexts, or to maintain enough information in some special data-structures to perform the split $C[E]$ using only local information and updates. Moreover, this "matching-modulo-the-CFG-of-evaluation-contexts" step needs to be done at every computation step during the execution of a program, so it may easily become the major bottleneck of an executable engine based on context reduction. Direct implementations of context reduction such as `PLT-Redex` cannot avoid paying a significant performance penalty, as the performance numbers in Table 15 of Section 11 show.

Context reduction is trickier to faithfully capture as a rewrite theory, since rewriting logic, by its locality, always applies a rule *in* the context, without actually having the capability of changing the given context. To faithfully model context-reduction, we make use of two equationally-defined operations: *s2c*, which splits a piece of syntax into a context and a redex, and *c2s*, which plugs a piece of syntax into a context. In our rewriting logic definition, $C[R]$ is *not a parsing convention*, but rather a *constructor* conveniently representing the pair (context C, redex R). In order to have an algebraic representation of contexts we extend the signature by adding a constant $[]$, representing the hole, for each syntactic category. The operation *s2c*, presented in Table 8, has an effect similar to what one achieves by parsing in context reduction, in the sense that given a piece of syntax it yields $C[R]$. The operation *c2s*, also presented in Table 8, is defined as a morphism on the syntax, but we get (from the defining equations) the guarantee that it will be applied only to "well-formed" contexts (i.e., contexts containing only one hole). The rewrite theory $\mathcal{R}_{CxtRed}$ is obtained by adding the rules in Table 9 to the equations of *s2c* and *c2s*.

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

Theorem 2 *Suppose that $s \simeq \sigma$. Then the following hold:*

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

Proof.

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

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

   Case a_1<=a_2: a_1 <= a_2 parses as a_1 <= $c[r]$ iff
   $a_1 \in Int$ and a_2 parses as $c[r]$ iff
   $a_1 \in Int$ and $\mathcal{R}_{CxtRed} \vdash s2c(a_2) = c[r]$ iff
   $\mathcal{R}_{CxtRed} \vdash s2c(a_1$<=$a_2) = (a_1$<=$c)[r]$.

   Case x:=a: x:=a parses as $[][x$:=$a]$ iff $a \in Int$, iff
   $\mathcal{R}_{CxtRed} \vdash s2c(x$:=$i) = [][x$:=$i]$.

   Case $st.a$: $st.a$ parses as $st.c[r]$ iff
   $st =$ `skip` and a parses as $c[r]$, iff
   $st =$ `skip` and $\mathcal{R}_{CxtRed} \vdash s2c(a) = c[r]$ iff
   $\mathcal{R}_{CxtRed} \vdash s2c(at.a) = st.c[r]$.

$\mathit{s2c}(\langle P, S\rangle) = \langle C, S\rangle[R]$	**if** $C[R] = \mathit{s2c}(P)$
$\mathit{s2c}(\texttt{skip}.I) = [\,][\texttt{skip}.I]$	
$\mathit{s2c}(\texttt{skip}.A) = (\texttt{skip}.C)[R]$	**if** $C[R] = \mathit{s2c}(A)$
$\mathit{s2c}(St.A) = (C.A)[R]$	**if** $C[R] = \mathit{s2c}(St)$
$\mathit{s2c}(\texttt{halt}\, I) = [\,][\texttt{halt}\, I]$	
$\mathit{s2c}(\texttt{halt}\, A) = (\texttt{halt}\, C)[R]$	**if** $C[R] = \mathit{s2c}(A)$
$\mathit{s2c}(\texttt{while}\, B\ St) = [\,][\texttt{while}\, B\ St]$	
$\mathit{s2c}(\texttt{if}\, T\, \texttt{then}\, St_1\, \texttt{else}\, St_2) = [\,][\texttt{if}\, T\, \texttt{then}\, St_1\, \texttt{else}\, St_2]$	
$\mathit{s2c}(\texttt{if}\, B\, \texttt{then}\, St_1\, \texttt{else}\, St_2) = (\texttt{if}\, C\, \texttt{then}\, St_1\, \texttt{else}\, St_2)[R]$	**if** $C[R] = \mathit{s2c}(B)$
$\mathit{s2c}(\{St\}) = [\,][\{St\}]$	
$\mathit{s2c}(\texttt{skip}; St_2) = [\,][\texttt{skip}; St_2]$	
$\mathit{s2c}(St_1; St_2) = (C; St_2)[R]$	**if** $C[R] = \mathit{s2c}(St_1)$
$\mathit{s2c}(X\texttt{:=}I) = [\,][X\texttt{:=}I]$	
$\mathit{s2c}(X\texttt{:=}A) = (X\texttt{:=}C)[R]$	**if** $C[R] = \mathit{s2c}(A)$
$\mathit{s2c}(I_1\texttt{<=}I_1) = [\,][I_1\texttt{<=}I_2]$	
$\mathit{s2c}(I\texttt{<=}A) = (I\texttt{<=}C)[R]$	**if** $C[R] = \mathit{s2c}(A)$
$\mathit{s2c}(A_1\texttt{<=}A_2) = (C\texttt{<=}A_2)[R]$	**if** $C[R] = \mathit{s2c}(A_1)$
$\mathit{s2c}(T\, \texttt{and}\, B_2) = [\,][T\, \texttt{and}\, B_2]$	
$\mathit{s2c}(B_1\, \texttt{and}\, B_2) = (C\, \texttt{and}\, B_2)[R]$	**if** $C[R] = \mathit{s2c}(B_1)$
$\mathit{s2c}(\texttt{not}\, T) = [\,][\texttt{not}\, T]$	
$\mathit{s2c}(\texttt{not}\, B) = (\texttt{not}\, C)[R]$	**if** $C[R] = \mathit{s2c}(B)$
$\mathit{s2c}(X) = [\,][X]$	
$\mathit{s2c}(\texttt{++}X) = [\,][\texttt{++}X]$	
$\mathit{s2c}(I_1 + I_2) = [\,][I_1 + I_2]$	
$\mathit{s2c}(I + A) = (I + C)[R]$	**if** $C[R] = \mathit{s2c}(A)$
$\mathit{s2c}(A_1 + A_2) = (C + A_2)[R]$	**if** $C[R] = \mathit{s2c}(A_1)$
$\mathit{c2s}([\,][H]) = H$	
$\mathit{c2s}(\langle P, S\rangle[H]) = \langle \mathit{c2s}(P[H]), S\rangle$	
$\mathit{c2s}(\langle I\rangle[H]) = \langle I\rangle$	
$\mathit{c2s}(E_1.E_2[H]) = \mathit{c2s}(E_1[H]).\mathit{c2s}(E_2[H])$	
$\mathit{c2s}(\texttt{halt}\, E[H]) = \texttt{halt}\, \mathit{c2s}(E[H])$	
$\mathit{c2s}(\texttt{while}\, E_1\ E_2[H]) = \texttt{while}\, \mathit{c2s}(E_1[H])\ \mathit{c2s}(E_2[H])$	
$\mathit{c2s}(\texttt{if}\, E\, \texttt{then}\, E_1\, \texttt{else}\, E_2[H]) = \texttt{if}\, \mathit{c2s}(E[H])\, \texttt{then}\, \mathit{c2s}(E_1[H])\, \texttt{else}\, \mathit{c2s}(E_2[H])$	
$\mathit{c2s}(\{E\}[H]) = \{\mathit{c2s}(E[H])\}$	
$\mathit{c2s}(E_1; E_2[H]) = \mathit{c2s}(E_1[H]); \mathit{c2s}(E_2[H])$	
$\mathit{c2s}(X\texttt{:=}E[H]) = X\texttt{:=}\mathit{c2s}(E[H])$	
$\mathit{c2s}(\texttt{skip}[H]) = \texttt{skip}$	
$\mathit{c2s}(E_1\texttt{<=}E_2[H]) = \mathit{c2s}(E_1[H])\texttt{<=}\mathit{c2s}(E_2[H])$	
$\mathit{c2s}(E_1\, \texttt{and}\, E_2[H]) = \mathit{c2s}(E_1[H])\, \texttt{and}\, \mathit{c2s}(E_2[H])$	
$\mathit{c2s}(\texttt{not}\, E[H]) = \texttt{not}\, \mathit{c2s}(E[H])$	
$\mathit{c2s}(\texttt{true}[H]) = \texttt{true}$	
$\mathit{c2s}(\texttt{false}[H]) = \texttt{false}$	
$\mathit{c2s}(\texttt{++}X[H]) = \texttt{++}X$	
$\mathit{c2s}(E_1 + E_2[H]) = \mathit{c2s}(E_1[H]) + \mathit{c2s}(E_2[H])$	
$\mathit{c2s}(I[H]) = I$	

Table 8: Equational definitions of *s2c* and *c2s*

$\cdot(I_1 + I_2) \to (I_1 +_{Int} I_2)$		
$\cdot(\langle P,S\rangle[X]) \to \langle P,S\rangle[(S[X])]$		
$\cdot(\langle P,S\rangle[\texttt{++}X]) \to \langle P,S[X \leftarrow I]\rangle[I]$	**if**	$I = s(S[X])$
$\cdot(I_1\texttt{<=}I_2) \to (I_1 \leq_{Int} I_2)$		
$\cdot(\texttt{true and } B) \to B$		
$\cdot(\texttt{false and } B) \to \texttt{false}$		
$\cdot(\texttt{not true}) \to \texttt{false}$		
$\cdot(\texttt{not false}) \to \texttt{true}$		
$\cdot(\texttt{if true then } St_1 \texttt{ else } St_2) \to St_1$		
$\cdot(\texttt{if false then } St_1 \texttt{ else } St_2) \to St_2$		
$\cdot(\texttt{skip}; St) \to St$		
$\cdot(\{St\}) \to St$		
$\cdot(\langle P,S\rangle[X\texttt{:=}I]) \to \langle P,S[X \leftarrow I]\rangle[\texttt{skip}]$		
$\cdot(\texttt{while } B\ St) \to \texttt{if } B \texttt{ then } (St;\texttt{while } B\ St) \texttt{ else skip}$		
$\cdot(C[\texttt{halt } I]) \to \langle I\rangle[[]]$		
$\cdot(C[\texttt{skip}.I]) \to \langle I\rangle[[]]$		
$\cdot(C[R]) \to C[R']$	**if**	$\cdot(R) \to R'$
$\cdot(\mathit{Cfg}) \to \mathit{c2s}(C[R])$	**if**	$\cdot(\mathit{s2c}(\mathit{Cfg})) \to C[R]$
$\mathit{eval}(P) = \mathit{reduction}(\langle P, \emptyset\rangle)$		
$\mathit{reduction}(\mathit{Cfg}) = \mathit{reduction}(\cdot(\mathit{Cfg}'))$		
$\mathit{reduction}(\langle I\rangle) = I$		

Table 9: $\mathcal{R}_{CxtRed}$ rewriting logic theory

Case $\langle p,\sigma\rangle$: $\langle p,\sigma\rangle$ parses as $c[r]$ iff
p parses as $c'[r]$ and $c = \langle c', s\rangle$ iff
$\mathcal{R}_{CxtRed} \vdash \mathit{s2c}(p) = c'[r]$ and $c = \langle c', s\rangle$ iff
$\mathcal{R}_{CxtRed} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c', s\rangle[r]$.

2. From the way it was defined, *c2s* acts as a morphism on the structure of syntactic constructs, changing $[]$ in C by R. Since *c2s* is defined for all constructors, it will work for any valid context C and pluggable expression e. Note, however, that *c2s* works as stated also on multi-contexts (i.e., on contexts with multiple holes), but this aspect does not interest us here.

3. There are several cases again to analyze, depending on the particular reduction that provoked the derivation $\mathit{CxtRed} \vdash \langle p,\sigma\rangle \to \langle p',\sigma\rangle$. We only discuss some cases; the others are treated similarly.

   $\mathit{CxtRed} \vdash \langle p,\sigma\rangle \to \langle p',\sigma'\rangle$ because of $\mathit{CxtRed} \vdash \langle c,\sigma\rangle[x] \to \langle c,\sigma\rangle[\sigma(x)]$ iff
   $\langle p,\sigma\rangle$ parses as $\langle c,\sigma\rangle[x]$ and $\langle p',\sigma'\rangle$ is $\langle c,\sigma\rangle[\sigma(x)]$ (in particular $\sigma' = \sigma$) iff
   $\mathcal{R}_{CxtRed} \vdash \mathit{s2c}(\langle p,s\rangle) = \langle c,s\rangle[x]$, $\mathcal{R}_{CxtRed} \vdash s[x] = i$ where $i = \sigma(x)$ and $\mathcal{R}_{CxtRed} \vdash \mathit{c2s}(\langle c,s\rangle[i]) = \langle p',s\rangle$ iff
   $\mathcal{R}_{CxtRed} \vdash \cdot(\langle p,s\rangle) \to^1 \langle p',s\rangle$, because $\mathcal{R}_{CxtRed} \vdash \cdot(\langle c,s\rangle[x]) \to^1 \langle c,s\rangle[i]$.

   $\mathit{CxtRed} \vdash \langle p,\sigma\rangle \to \langle p',\sigma\rangle$ because of $\frac{\texttt{not true}\to\texttt{false}}{c[\texttt{not true}]\to c[\texttt{false}]}$ for some evaluation context c iff
   $\langle p,\sigma\rangle$ parses as $c[\texttt{not true}]$ and $\langle p',\sigma\rangle$ is $c[\texttt{false}]$ iff
   $\mathcal{R}_{CxtRed} \vdash \mathit{s2c}(\langle p,s\rangle) = c[\texttt{not true}]$ and $\mathcal{R}_{CxtRed} \vdash \mathit{c2s}(c[\texttt{false}]) = \langle p',s\rangle$ iff
   $\mathcal{R}_{CxtRed} \vdash \cdot(\langle p,s\rangle) \to^1 \langle p',s\rangle$, because $\mathcal{R}_{CxtRed} \vdash \cdot(c[\texttt{not true}]) \to^1 c[\texttt{false}]$ (which follows since $\mathcal{R}_{CxtRed} \vdash \cdot(\texttt{not true}) \to^1 \texttt{false}$).

   $\mathit{CxtRed} \vdash \langle p,\sigma\rangle \to \langle p',\sigma'\rangle$ because of $\mathit{CxtRed} \vdash \langle c,\sigma\rangle[x\texttt{:=}i] \to \langle c,\sigma[i/x][\texttt{skip}]\rangle$ iff
   $\langle p,\sigma\rangle$ parses as $\langle c,\sigma\rangle[x\texttt{:=}i]$, $\sigma' = \sigma[i/x]$ and $\langle p',\sigma'\rangle$ is $\langle c,\sigma'\rangle[\texttt{skip}]$ iff

$\mathcal{R}_{CxtRed} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c, s\rangle[x\texttt{:=}i]$, $s' = s[x \leftarrow i] \simeq \sigma'$ and $\mathcal{R}_{CxtRed} \vdash \mathit{c2s}(\langle c, s'\rangle[\texttt{skip}]) = \langle p', s'\rangle$ iff $\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s\rangle) \rightarrow^1 \langle p', s'\rangle$, because $\mathcal{R}_{CxtRed} \vdash \cdot(\langle c, s\rangle[x\texttt{:=}i]) \rightarrow^1 \langle c, s'\rangle[\texttt{skip}]$.

4. $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \rightarrow \langle i\rangle$ because of $\mathit{CxtRed} \vdash c[\texttt{skip.}i] \rightarrow \langle i\rangle$ iff
$\langle p, \sigma\rangle$ parses as $\langle [], \sigma\rangle[\texttt{skip.}i]$ iff
$\mathcal{R}_{CxtRed} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle [], s\rangle[\texttt{skip.}i]$ iff
$\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s\rangle) = \langle i\rangle$, since $\mathcal{R}_{CxtRed} \vdash \cdot(\langle [], \sigma\rangle[\texttt{skip.}i]) \rightarrow^1 \langle i\rangle[[]]$ and since $\mathcal{R}_{CxtRed} \vdash \mathit{c2s}(\langle i\rangle[[]]) = \langle i\rangle$.

   Also, $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \rightarrow \langle i\rangle$ because of $\mathit{CxtRed} \vdash c[\texttt{halt}\ \ i] \rightarrow \langle i\rangle$ iff
$\langle p, \sigma\rangle$ parses as $\langle c, \sigma\rangle[\texttt{halt}\ \ i]$ iff
$\mathcal{R}_{CxtRed} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c, s\rangle[\texttt{halt}\ \ i]$ iff
$\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s\rangle) = \langle i\rangle$ since $\mathcal{R}_{CxtRed} \vdash \cdot(\langle c, \sigma\rangle[\texttt{halt}\ \ i]) \rightarrow^1 \langle i\rangle[[]]$ and since $\mathcal{R}_{CxtRed} \vdash \mathit{c2s}(\langle i\rangle[[]]) = \langle i\rangle$.

5. This part of the proof follows the same pattern as that for the similar property for *SmallStep* (Proposition 3), using the above properties and replacing *smallstep* by *reduction*.

□

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

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

9 A Continuation-Based Semantics

The idea of continuation-based interpreters for programming languages and their relation to abstract machines has been well studied (see, for example, [32]). In this section we propose a rewriting logic theory based on a structure that provides a *first-order* representation of continuations; this is the only reason why we call this structure a "continuation"; but notice that it can just as well be regarded as a post-order representation of the abstract syntax tree of the program, so one needs no prior knowledge of continuations [32] in order to understand this section. We will show the equivalence of this theory to the context reduction semantics theory.

Based on the desired order of evaluation, the program is sequentialized by transforming it into a list of tasks to be performed in order. This is done once and for all at the beginning, the benefit being that at any subsequent moment in time we know precisely where the next redex is: at the top of the tasks list. We call this list of tasks a *continuation* because it resembles the idea of continuations as higher-order functions. However, our continuation is a pure first order flattening of the program. For example $\mathit{aexp}(A_1 + A_2) = (\mathit{aexp}(A_1), \mathit{aexp}(A_2)) \curvearrowright +$ precisely encodes the order of evaluation: first A_1, then A_2, then sum the values. Also, $\mathit{stmt}(\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2) = B \curvearrowright \mathit{if}(\mathit{stmt}(St_1), \mathit{stmt}(St_2))$ says that St_1 and St_2 are dependent on the value of B for their evaluation.

The top level configuration is constructed by an operator "__" putting together the store (wrapped by a constructor *store*) and the continuation (wrapped by k). Also, syntax is added for the continuation items. Here the distinction between equations and rules becomes even more obvious: equations are used to prepare the context in which a computation step can be applied, while rewrite rules exactly encode the computation

$\mathit{aexp}(I) = I$
$\mathit{aexp}(A_1 + A_2) = (\mathit{aexp}(A_1), \mathit{aexp}(A_2)) \curvearrowright +$
$k(\mathit{aexp}(X) \curvearrowright K)\ \mathit{store}(\mathit{Store}) \to k(\mathit{Store}[X] \curvearrowright K)\ \mathit{store}(\mathit{Store})$
$k(\mathit{aexp}(\texttt{++}X) \curvearrowright K)\ \mathit{store}((X = I)\,\mathit{Store}) \to k(s(I) \curvearrowright K)\ \mathit{store}((X = s(I))\,\mathit{Store})$
$.\ k(I_1, I_2 \curvearrowright + \curvearrowright K) \to k(I_1 +_{\mathit{Int}} I_2 \curvearrowright K)$

$\mathit{bexp}(\mathit{true}) = \mathit{true}$
$\mathit{bexp}(\mathit{false}) = \mathit{false}$
$\mathit{bexp}(A_1\texttt{<=}A_2) = (\mathit{aexp}(A_1), \mathit{aexp}(A_2)) \curvearrowright \leq$
$\mathit{bexp}(B_1\ \texttt{and}\ B_2) = \mathit{bexp}(B_1) \curvearrowright \mathit{and}(\mathit{bexp}(B_2))$
$\mathit{bexp}(\texttt{not}\ B) = \mathit{bexp}(B) \curvearrowright \mathit{not}$
$k(I_1, I_2 \curvearrowright \leq \curvearrowright K) \to k(I_1 \leq_{\mathit{Int}} I_2 \curvearrowright K)$
$k(\mathit{true} \curvearrowright \mathit{and}(K_2) \curvearrowright K) \to k(K_2 \curvearrowright K)$
$k(\mathit{false} \curvearrowright \mathit{and}(K_2) \curvearrowright K) \to k(\mathit{false} \curvearrowright K)$
$k(T \curvearrowright \mathit{not} \curvearrowright K) \to k(\mathit{not}_{\mathit{Bool}} T \curvearrowright K)$

$\mathit{stmt}(\texttt{skip}) = \mathit{nothing}$
$\mathit{stmt}(X := A) = \mathit{aexp}(A) \curvearrowright \mathit{write}(X)$
$\mathit{stmt}(\mathit{St}_1; \mathit{St}_2) = \mathit{stmt}(\mathit{St}_1) \curvearrowright \mathit{stmt}(\mathit{St}_2)$
$\mathit{stmt}(\{\mathit{St}\}) = \mathit{stmt}(\mathit{St})$
$\mathit{stmt}(\texttt{if}\ B\ \texttt{then}\ \mathit{St}_1\ \texttt{else}\ \mathit{St}_2) = \mathit{bexp}(B) \curvearrowright \mathit{if}(\mathit{stmt}(\mathit{St}_1), \mathit{stmt}(\mathit{St}_2))$
$\mathit{stmt}(\texttt{while}\ B\ \mathit{St}) = \mathit{bexp}(B) \curvearrowright \mathit{while}(\mathit{bexp}(B), \mathit{stmt}(\mathit{St}))$
$\mathit{stmt}(\texttt{halt}\ A) = \mathit{aexp}(A) \curvearrowright \mathit{halt}$
$k(I \curvearrowright \mathit{write}(X) \curvearrowright K)\ \mathit{store}(\mathit{Store}) \to k(K)\ \mathit{store}(\mathit{Store}[X \leftarrow I])$
$k(\mathit{true} \curvearrowright \mathit{if}(K_1, K_2) \curvearrowright K) \to k(K_1 \curvearrowright K)$
$k(\mathit{false} \curvearrowright \mathit{if}(K_1, K_2) \curvearrowright K) \to k(K_2 \curvearrowright K)$
$k(\mathit{true} \curvearrowright \mathit{while}(K_1, K_2) \curvearrowright K) \to k(K_2 \curvearrowright K_1 \curvearrowright \mathit{while}(K_1, K_2) \curvearrowright K)$
$k(\mathit{false} \curvearrowright \mathit{while}(K_1, K_2) \curvearrowright K) \to k(K)$
$k(I \curvearrowright \mathit{halt} \curvearrowright K) \to k(I)$

$\mathit{pgm}(\mathit{St}.A) = \mathit{stmt}(\mathit{St}) \curvearrowright \mathit{aexp}(A)$

$\langle P \rangle = \mathit{result}(k(\mathit{pgm}(P))\ \mathit{store}(\mathit{empty}))$
$\mathit{result}(k(I)\ \mathit{store}(\mathit{Store})) = I$

using the (equationally defined) mechanism for evaluating lists of expressions:

$$k((Vl, Ke, Kel) \curvearrowright K) = k(Ke \curvearrowright (Vl, \mathit{nothing}, Kel) \curvearrowright K)$$

Note. Because in rewriting engines equations are also executed by rewriting, one would need to split the rule for evaluating expressions into two rules:

$$k((Vl, Ke, Kel) \curvearrowright K) = k(Ke \curvearrowright (Vl, \mathit{nothing}, Kel) \curvearrowright K)$$
$$k(V \curvearrowright (Vl, \mathit{nothing}, Kel) \curvearrowright K) = k((Vl, V, Kel) \curvearrowright K)$$

Table 10: Rewriting logic theory $\mathcal{R}_K$ (continuation-based definition of the language)

steps semantically, yielding the intended computational granularity. Specifically *pgm*, *stmt*, *bexp*, *aexp* are used to flatten the program to a continuation, taking into account the order of evaluation. The continuation is defined as a list of tasks, where the list constructor "$\_ \curvearrowright \_$" is associative, having as identity a constant "*nothing*". We also use lists of values and continuations, each having an associative list append constructor "$\_, \_$" with identity ".". We use variables K and V to denote continuations and values, respectively; also, we use Kl and Vl for lists of continuations and values, respectively. The rewrite theory $\mathcal{R}_K$ specifying the continuation-based definition of our example language is given in Table 10.

The most important benefit of this transformation is that of gaining locality. Now one needs to specify from the context only what is needed to perform the computation. This indeed gives the possibility of achieving "true concurrency", since rules which do not act on the same parts of the context can be applied in parallel. We here only discuss the sequential variant of our continuation-based semantics, because our language is sequential. In [71] we show how the same technique can be used, with no additional effort, to define concurrent languages; the idea is, as expected, that one continuation structure is generated for each concurrent thread or process. Then rewrite rules can apply "truly concurrently" at the tops of continuations.

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

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

9.1 Relation with Context Reduction

We next show the equivalence between the continuation-based and the context reduction rewriting logic definitions. The specification in Table 11 relates the two semantics, showing that at each computational "point" it is possible to extract from our continuation structure the current expression being evaluated. For each syntactical construct $Syn \in \{AExp, BExp, Stmt, Pgm\}$, we equationally define two (partial) functions:

- *k2Syn* takes a continuation encoding of *Syn* into *Syn*; and
- *kSyn* extracts from the tail of a continuation a *Syn* and returns it together with the remaining prefix continuation.

Together, these two functions can be regarded as a parsing process, where the continuation plays the role of "unparsed" syntax, while *Syn* is the abstract syntax tree, i.e., the "parsed" syntax. The formal definitions of *k2Syn* and *kSyn* are given in Table 11.

We will show below that for any step *CxtRed* does, $\mathcal{R}_K$ performs at most one step to reach the same[5] configuration. No steps are performed for `skip`, or for dissolving a block (because these were dealt with when we transformed the syntax into continuation form), or for dissolving a statement into a skip (there is no need for that when using continuations). Also, no steps will be performed for loop unrolling, because this is *not* a computational step; it is a straightforward structural equivalence. In fact, note that, because of its incapacity to distinguish between computational steps and structural equivalences, *CxtRed* does not

[4]However, we regard this as minor syntactic details. After all, the program needs to be transformed into an abstract syntax tree (AST) in any of the previous formalisms. Whether the AST is kept in prefix versus postfix order is somewhat irrelevant.

[5]"same" modulo irrelevant but equivalent syntactic notational conventions.

$\mathit{k2Pgm}(K) = \mathit{k2Stmt}(K').A$ **if** $\{K', A\} = \mathit{kAExp}(K)$

$\mathit{k2Stmt}(\mathit{nothing}) = \texttt{skip}$

$\mathit{k2Stmt}(K) = \mathit{k2Stmt}(K'); \mathit{St}$ **if** $\{K', \mathit{St}\} = \mathit{kStmt}(K) \wedge K' \neq \mathit{nothing}$

$\mathit{k2Stmt}(K) = \mathit{St}$ **if** $\{K', \mathit{St}\} = \mathit{kStmt}(K) \wedge K' = \mathit{nothing}$

$\mathit{kStmt}(K \curvearrowright \mathit{write}(X)) = \{K', X \texttt{:=} A\}$ **if** $\{K', A\} = \mathit{kAExp}(K)$

$\mathit{kStmt}(K \curvearrowright \mathit{while}(K_1, K_2)) = \{K', \texttt{if}\ B\ \texttt{then}\ \{\mathit{St}; \texttt{while}\ B_1 \mathit{St}\}\ \texttt{else skip}\}$
    if $\{K', B\} = \mathit{kBExp}(K) \wedge B_1 = \mathit{k2BExp}(K_1) \wedge \mathit{St} = \mathit{k2Stmt}(K_2) \wedge B \neq B_1$

$\mathit{kStmt}(K \curvearrowright \mathit{while}(K_1, K_2)) = \{K', \texttt{while}\ B\ \mathit{St}\}$
    if $\{K', B\} = \mathit{kBExp}(K) \wedge B_1 = \mathit{k2BExp}(K_1) \wedge \mathit{St} = \mathit{k2Stmt}(K_2) \wedge B = B_1$

$\mathit{kStmt}(K \curvearrowright \mathit{if}(K_1, K_2)) = \{K', \texttt{if}\ B\ \texttt{then}\ \mathit{k2Stmt}(K_1)\ \texttt{else}\ \mathit{k2Stmt}(K_2)\}$
    if $\{K', B\} = \mathit{kBExp}(K)$

$\mathit{kStmt}(K \curvearrowright \mathit{halt}) = \{K', \texttt{halt}\ A\}$ **if** $\{K', A\} = \mathit{kAExp}(K)$

$\mathit{k2AExp}(K) = A$ **if** $\{\mathit{nothing}, A\} = \mathit{kAExp}(K)$

$\mathit{kAExp}(K \curvearrowright \mathit{kv}(Kl, Vl) \curvearrowright K') = \mathit{kAExp}(Vl, K, Kl \curvearrowright K')$

$\mathit{kAExp}(K \curvearrowright \mathit{aexp}(A)) = \{K, A\}$

$\mathit{kAExp}(K \curvearrowright I) = \{K, I\}$

$\mathit{kAExp}(K \curvearrowright K_1, K_2 \curvearrowright +) = \{K, \mathit{k2AExp}(K_1) + \mathit{k2AExp}(K_2)\}$

$\mathit{k2BExp}(K) = B$ **if** $\{\mathit{nothing}, B\} = \mathit{kBExp}(K)$

$\mathit{kBExp}(K \curvearrowright \mathit{kv}(Kl, Vl) \curvearrowright K') = \mathit{kBExp}(Vl, K, Kl \curvearrowright K')$

$\mathit{kBExp}(K \curvearrowright T) = \{K, T\}$

$\mathit{kBExp}(K \curvearrowright K_1, K_2 \curvearrowright \leq) = \{K, \mathit{k2AExp}(K_1)\texttt{<=}\mathit{k2AExp}(K_2)\}$

$\mathit{kBExp}(K \curvearrowright \mathit{and}(K_2)) = \{K_1, B_1\ \texttt{and}\ \mathit{k2BExp}(K_2)\}$ **if** $\{K_1, B_1\} = \mathit{kBExp}(K)$

$\mathit{kBExp}(K \curvearrowright \mathit{not}) = \{K', \texttt{not}\ B\}$ **if** $\{K', B\} = \mathit{kBExp}(K)$

Table 11: Recovering the abstract syntax trees from continuations

capture the intended granularity of `while`: it wastes a computation step for unrolling the loop and one when dissolving the while into skip; neither of these steps has any computational content.

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

$$k(\mathit{syn}(\mathit{Syn}) \curvearrowright K') = k(\mathit{syn}(\mathit{Syn}) \curvearrowright \mathit{syn}([]) \curvearrowright K'),$$

replacing the equation for evaluating lists of expressions, namely,

$$k((\mathit{Vl}, \mathit{Ke}, \mathit{Kel}) \curvearrowright K) = k(\mathit{Ke} \curvearrowright (\mathit{Vl}, \mathit{nothing}, \mathit{Kel}) \curvearrowright K),$$

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

$$k((\mathit{Vl}, \mathit{Ke}, \mathit{Kel}) \curvearrowright K) = k(\mathit{Ke} \curvearrowright (\mathit{Vl}, \mathit{syn}([]), \mathit{Kel}) \curvearrowright K)$$

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

Proposition 4 *For each arithmetic context c in CxtRed and $r \in$ AExp, we have that $\mathcal{R}_{K'} \vdash k(\mathit{aexp}(c[r])) = k(\mathit{aexp}(r) \curvearrowright \mathit{aexp}(c)))$. Similarly for any possible combination for c and r among AExp, BExp, Stmt, Pgm, Cfg.*

(Note that r in the proposition above need not be a redex, but can be any expression of the right syntactical category, i.e., pluggable in the hole.)
Proof.

$\texttt{++}x = [][\texttt{++}x]$: $\mathcal{R}_{K''} \vdash k(\mathit{aexp}(\texttt{++}x)) = k(\mathit{aexp}(\texttt{++}x) \curvearrowright \mathit{aexp}([]))$

$a_1 + a_2 = [] + a_2[a_1]$: $\mathcal{R}_{K''} \vdash k(\mathit{aexp}(a_1 + a_2)) = k((\mathit{aexp}(a_1), \mathit{aexp}(a_2)) \curvearrowright +)$
$\quad = k(\mathit{aexp}(a_1) \curvearrowright (\mathit{aexp}([]), \mathit{aexp}(a_2)) \curvearrowright +) = k(\mathit{aexp}(a_1) \curvearrowright \mathit{aexp}([] + a_2))$

$i_1 + a_2 = i_1 + [][a_2]$: $\mathcal{R}_{K''} \vdash k(\mathit{aexp}(i_1 + a_2)) = k((\mathit{aexp}(i_1), \mathit{aexp}(a_2)) \curvearrowright +)$
$\quad = k(\mathit{aexp}(a_2) \curvearrowright (i_1, \mathit{aexp}([])) \curvearrowright +) = k(\mathit{aexp}(a_2) \curvearrowright \mathit{aexp}(i_1 + [])).$

$b_1 \texttt{ and } b_2 = [] \texttt{ and } b_2[b_1]$: $\mathcal{R}_{K''} \vdash k(\mathit{bexp}(b_1 \texttt{ and } b_2)) = k(\mathit{bexp}(b_1) \curvearrowright \mathit{and}(\mathit{bexp}(b_2)))$
$\quad = k(\mathit{bexp}(b_1) \curvearrowright \mathit{bexp}([]) \curvearrowright \mathit{and}(\mathit{aexp}(b_2))) = k(\mathit{bexp}(b_1) \curvearrowright \mathit{bexp}([]\texttt{and } b_2)).$

$t \texttt{ and } b_2 = [][t \texttt{ and } b_2]$: $\mathcal{R}_{K''} \vdash k(\mathit{bexp}(t \texttt{ and } b_2)) = k(\mathit{bexp}(t \texttt{ and } b_2) \curvearrowright \mathit{bexp}([])).$

$st.a = [].a[st]$: $\mathcal{R}_{K''} \vdash k(\mathit{pgm}(st.a)) = k(\mathit{stmt}(st) \curvearrowright \mathit{aexp}(a))$
$\quad = k(\mathit{stmt}(st) \curvearrowright \mathit{stmt}([]) \curvearrowright \mathit{aexp}(a)) = k(\mathit{stmt}(st) \curvearrowright \mathit{pgm}([].a)).$

$\texttt{skip}.a = \texttt{skip}.[][a]$: $\mathcal{R}_{K''} \vdash k(\mathit{pgm}(\texttt{skip}.a)) = k(\mathit{stmt}(\mathit{skip}) \curvearrowright \mathit{aexp}(a))$
$\quad = k(\mathit{aexp}(a)) = k(\mathit{aexp}(a) \curvearrowright \mathit{aexp}([]))$
$\quad = k(\mathit{aexp}(a) \curvearrowright \mathit{stmt}(\mathit{skip}) \curvearrowright \mathit{aexp}([])) = k(\mathit{aexp}(a) \curvearrowright \mathit{pgm}(\texttt{skip}.[])).$

All other constructs are dealt with in a similar manner. □

Lemma 1 $\mathcal{R}_{K'} \vdash k(k_1) = k(k_2)$ *implies that for any k_{rest}, $\mathcal{R}_{K'} \vdash k(k_1 \curvearrowright k_{rest}) = k(k_2 \curvearrowright k_{rest})$*

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

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

Theorem 3 *Suppose* $s \simeq \sigma$.

1. *If* $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ *then* $\mathcal{R}_{K'} \vdash k(\mathit{pgm}(p))\ \mathit{store}(s) \to^{\leq 1} k(\mathit{pgm}(p'))\ \mathit{store}(s')$ *and* $s' \simeq \sigma'$, *where* $\to^{\leq 1} = \to^0 \cup \to^1$.

2. *If* $\mathcal{R}_{K'} \vdash k(\mathit{pgm}(p))\ \mathit{store}(s) \to k(k')\ \mathit{store}(s')$ *then there exists* p' *and* σ' *such that* $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to^* \langle p', \sigma'\rangle$, $\mathcal{R}_{K'} \vdash k(\mathit{pgm}(p')) = k(k')$ *and* $s' \simeq \sigma'$.

3. $\mathit{CxtRed} \vdash \langle p, \bot\rangle \to^* i$ *iff* $\mathcal{R}_{K'} \vdash \langle p\rangle \to i$ *for any* $p \in \mathit{Pgm}$ *and* $i \in \mathit{Int}$.

Proof. Sketch.

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

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

3. Using the previous two statements, and the rules for halt or end of the program from both definitions. We exemplify only halt, the end of the program is similar, but simpler. For $\langle c, \sigma\rangle[\texttt{halt}\ i] \to i$ we have $k(i \curvearrowright \mathit{halt} \curvearrowright k) \to k(i)$, and combined with $\mathcal{R}_{K'} \vdash \mathit{result}(k(i)\ \mathit{store}(s)) = i$ we obtain $\mathcal{R}_{K'} \vdash \mathit{result}(k(\mathit{pgm}(c[\texttt{halt}\ i]))\ \mathit{store}(s)) \to i$.

□

10 The Chemical Abstract Machine

Berry and Boudol's *chemical abstract machine*, or *Cham* [7], is both a model of concurrency and a specific style of giving operational semantics definitions. Properly speaking, it is not an SOS definitional style. Berry and Boudol identify a number of limitations inherent in SOS, particularly its lack of true concurrency, and what might be called SOS's rigidity and slavery to syntax [7]. They then present the Cham as an *alternative* to SOS. In fact, as already pointed out in [48], what the Cham is, is a particular definitional style *within* RLS. That is, every Cham *is*, by definition, a specific kind of rewrite theory; and Cham computation is precisely concurrent rewriting computation; that is, proof in rewriting logic.

The basic metaphor giving its name to the Cham is inspired by Banâtre and Le Mètayer's GAMMA language [5]. It views a distributed state as a "solution" in which many "molecules" float, and understands concurrent transitions as "reactions" that can occur simultaneously in many points of the solution. It is possible to define a variety of chemical abstract machines. Each of them corresponds to a rewrite theory satisfying certain common conditions.

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

sorts Molecule, Molecules, Solution .
subsorts Solution $<$ *Molecule* $<$ *Molecules .*
op $\lambda :\longrightarrow$*Molecules .*
op $\_ , \_ :$ *Molecules Molecules*$\longrightarrow$*Molecules .*
op $\{\!|\_|\!\}$: *Molecules*$\longrightarrow$*Solution* . `*** membrane operator`
op $\_ \lhd \_ :$ *Molecule Solution*$\longrightarrow$*Molecule* . `*** airlock operator`

A *Cham* is then a rewrite theory $\mathcal{C} = (\Sigma, AC, R)$, with $\Sigma \supseteq \Omega$, together with a partition $R = \mathit{Reaction} \uplus \mathit{Heating} \uplus \mathit{Cooling} \uplus \mathit{AirlockAx}$. The associativity and commutativity (AC) axioms are asserted of the operator $\_ , \_$, which has identity λ. The rules in R may involve variables, but are subject to certain syntactic restrictions that guarantee an efficient form of AC matching [7]. *AirlockAx* is the bidirectional rule[6] $\{\!|m, M|\!\} \rightleftharpoons \{\!|m \rhd \{\!|M|\!\}|\!\}$, where m is a variable of sort *Molecule* and M a variable of sort *Molecules*. The purpose of this axiom is to choose one of the molecules m in a solution as a candidate for reaction with other molecules outside its membrane. The *Heating* and *Cooling* rules can typically be paired, with each rule $t \longrightarrow t' \in \mathit{Heating}$ having a symmetric rule $t' \longrightarrow t \in \mathit{Cooling}$, and vice-versa, so that we can view them as a single set of bidirectional rules $t' \rightleftharpoons t$ in *Heating-Cooling*.

Berry and Boudol [7] make a distinction between *rules*, which are rewrite rules specific to each Cham—and consist of the *Reaction*, *Heating*, and *Cooling* rules—and *laws* which are general properties applying to all Chams for governing the admissible transitions. The first three laws, the *Reaction*, *Chemical* and *Membrane* laws, just say that the Cham evolves by AC-rewriting. The fourth law states the axiom *AirlockAx*. The *Reaction* rules are the heart of the Cham and properly correspond to state transitions. The rules in *Heating-Cooling* express *structural equivalence*, so that the *Reaction* rules may apply after the appropriate structurally equivalent syntactic form is found. A certain strategy is typically given to address the problem of finding the right structural form, for example to perform "heating" as much as possible. In rewriting logic terms, a more abstract alternative view it to regard each Cham as a rewrite theory $\mathcal{C} = (\Sigma, ACI \cup \mathit{Heating\text{-}Cooling} \cup \mathit{AirlockAx}, \mathit{Reaction})$, in which the *Heating-Cooling* rules and the *AirlockAx* axiom have been made part of the theory's equational axioms. That is, we can more abstractly view the *Reaction* rules as applied *modulo* $ACI \cup \mathit{Heating\text{-}Cooling} \cup \mathit{AirlockAx}$.

As Berry and Boudol demonstrate in [7], the Cham is particularly well-suited to give semantics to concurrent calculi, yielding considerably simpler definitions than those afforded by SOS. In particular, [7] presents semantic definitions for the TCCS variant of CCS, a concurrent λ-calculus, and Milner's π-calculus. Milner himself also used Cham ideas to provide a compact formulation of his π-calculus [57]. Since our example language is sequential, it cannot take full advantage of the Cham's true concurrent capabilities. Nevertheless, there are interesting Cham features that, as we explain below, turn out to be useful even in this sequential language application. A Cham semantics for our language is given in Table 12. Note that, since the Cham is itself a rewrite theory, in this case there is no need for a representation in RLS, nor for a proof of correctness of such a representation; that is, the "representational distance" in this case is equal to 0. Again, RLS does not advocate any particular definitional style: the Cham style is just one possibility among many, having its own advantages and limitations.

The *CHAM* definition for our simple programming language takes the *CxtRed* definition in Table 7 as a starting point. We distinguish two kinds of molecules: syntactic molecules and store molecules. Syntactic molecules are either evaluation contexts or redexes. Store molecules are pairs (x, i), where x is a variable and i is an integer. The store is a solution containing store molecules. Definitions of evaluation contexts are translated into heating and cooling rules, bringing the redex to the top of the solution. This allows for the reduction rules to only operate at the top, in a way somehow similar to how it is done for continuation based definitions in Table 10.

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

[6]Which is of course understood as a pair of rules, one in each direction.

$$CxtRed \vdash \langle P, \sigma \rangle \to \langle P', \sigma' \rangle \iff CHAM \vdash \{|P|\}\ \{|\sigma|\} \rightharpoonup^{*}; \to^{1}; \leftharpoondown^{*} \{|P'|\}\ \{|\sigma'|\}$$

Strengths. Being a special case of rewriting logic, it inherits many of benefits of rewriting logic, being specially well-suited for describing truly concurrent computations and concurrent calculi.

Weaknesses. Heating/cooling rules are hard to implement efficiently – an implementation allowing them to be bidirectional in an uncontrolled manner would have to *search* for solutions, possibly leading to a combinatorial explosion. Rewriting strategies such as those in [9, 90, 27] can be of help for solving particular instances of this problem.

11 Experiments

RLS specifications can mechanically be turned into interpreters for the specified programming language. To analyze the efficiency of this approach, we wrote the RLS definitions of the language presented above in two rewrite engines, `ASF+SDF 1.5` (a compiler) and `Maude 2.2` (a fast interpreter with good tool support), and several programming languages with built-in support for matching, i.e., Haskell, Ocaml and Prolog. For each definitional style tested (except small-step SOS), we have included for comparison interpreters following that definitional style in Scheme, modifying existing interpreters used to teach programming languages. For Scheme we have adapted the definitions from [33], chapter 3.9 (evaluation semantics) and 7.3 (continuation based semantics) and a PLT-Redex definition given as example in the installation package (for context reduction). Big-step definitions are also compared against bc, a C-written interpreter for a subset of C working only with integers and two interpreters defined using monads in Haskell and Ocaml. Since RLS representations of MSOS and Cham definitions rely intensively on matching modulo associativity and commutativity, which is only supported by Maude, we have performed no experiments for them.

One of the programs chosen to test various implementations consists of n nested loops, each of 2 iterations, parameterized by n. The other program is verifying the Collatz' conjecture up to 300 (using repeated subtraction to compute division). The following tables give for each definitional style the running time of the various interpreters. For the largest number n (18) of nested loops, peak memory usage was also recorded. Times are expressed in seconds. A limit of 700mb was set on memory usage, to avoid swapping; the symbol "-" found in a table cell signifies that the memory limit was reached. For Haskell we have used the `ghc` compiler. For Ocaml we have used the `ocamlcopt` compiler. For Prolog we have compiled the programs using `gprolog` compiler. For Scheme we have used the PLT-Scheme interpreter. Tests were performed on an Intel Pentium 4@2GHz with 1GB RAM, running Linux.

Prolog yields pretty fast interpreters. However, for backtracking reasons, it needs to maintain the stack of all predicates tried on the current path, thus the amount of memory grows with the number of computational steps. The style promoted in [33] seems to also take into account efficiency. The only drawback is the fact that it looks more like an interpreter of a big-step definition, the representational distance to the big-step definition being much bigger than in interpreters based on RLS. The PLT-Redex implementation of context reduction seems to serve more a didactic purpose. It compensates lack of speed by providing a nice interface and the possibility to visually trace a run. The rewriting logic implementations seem to be quite efficient in terms of speed and memory usage, while keeping a minimal representational distance to the operational semantics definitions. In particular, RLS definitions interpreted in Maude are comparable in terms of efficiency with the interpreters in Scheme, while having the advantage of being formal definitions. Also, it is good to notice that compiled versions of RLS definitions can reach the speed of the hand-optimized, C-written `bc` interpreter.

12 Related Work

There is much related work on frameworks for defining programming languages. Without trying to be exhaustive, we mention some of them. We do not try to give detailed comparisons with each approach, but

$\{\!|St.A|\!\} \rightleftharpoons \{\!|St \rhd \{\!|[].A|\!\}|\!\}$
$\{\!|\texttt{skip}.A|\!\} \rightleftharpoons \{\!|A \rhd \{\!|\texttt{skip}.[]|\!\}|\!\}$
$\{\!|X\texttt{:=}A \rhd C|\!\} \rightleftharpoons \{\!|A \rhd \{\!|X\texttt{:=}[] \rhd C|\!\}|\!\}$
$\{\!|St_1; St_2 \rhd C|\!\} \rightleftharpoons \{\!|St_1 \rhd \{\!|[]; St_2 \rhd C|\!\}|\!\}$
$\{\!|\texttt{if}\, B \,\texttt{then}\, St_1 \,\texttt{else}\, St_2 \rhd C|\!\} \rightleftharpoons \{\!|B \rhd \{\!|\texttt{if}\, [] \,\texttt{then}\, St_1 \,\texttt{else}\, St_2 \rhd C|\!\}|\!\}$
$\{\!|\texttt{halt}\, A \rhd C|\!\} \rightleftharpoons \{\!|A \rhd \{\!|\texttt{halt}\, [] \rhd C|\!\}|\!\}$
$\{\!|A_1\texttt{<=}A_2 \rhd C|\!\} \rightleftharpoons \{\!|A_1 \rhd \{\!|[]\texttt{<=}A_2 \rhd C|\!\}|\!\}$
$\{\!|I\texttt{<=}A \rhd C|\!\} \rightleftharpoons \{\!|A \rhd \{\!|I\texttt{<=}[] \rhd C|\!\}|\!\}$
$\{\!|B_1 \,\texttt{and}\, B_2 \rhd C|\!\} \rightleftharpoons \{\!|B_1 \rhd \{\!|[] \,\texttt{and}\, B_2 \rhd C|\!\}|\!\}$
$\{\!|\texttt{not}\, B \rhd C|\!\} \rightleftharpoons \{\!|B \rhd \{\!|\texttt{not}\, [] \rhd C|\!\}|\!\}$
$\{\!|A_1 + A_2 \rhd C|\!\} \rightleftharpoons \{\!|A_1 \rhd \{\!|[] + A_2 \rhd C|\!\}|\!\}$
$\{\!|I + A \rhd C|\!\} \rightleftharpoons \{\!|A \rhd \{\!|I + [] \rhd C|\!\}|\!\}$

$\{\!|I_1 + I_2 \rhd C|\!\} \to \{\!|(I_1 +_{Int} I_2)|\!\} \rhd C$
$\{\!|X \rhd C|\!\}, \{\!|(X, I) \rhd \sigma|\!\} \to \{\!|I \rhd C|\!\}, \{\!|(X, I) \rhd \sigma|\!\}$
$\{\!|\texttt{++}X \rhd C|\!\}, \{\!|(X, I) \rhd \sigma|\!\} \to \{\!|I +_{Int} 1 \rhd C|\!\}, \{\!|(X, I +_{Int} 1) \rhd \sigma|\!\}$

$\{\!|I_1\texttt{<=}I_2 \rhd C|\!\} \to \{\!|(I_1 \leq_{Int} I_2)|\!\} \rhd C$
$\{\!|\texttt{true}\,\texttt{and}\, B \rhd C|\!\} \to \{\!|B \rhd C|\!\}$
$\{\!|\texttt{false}\,\texttt{and}\, B \rhd C|\!\} \to \{\!|\texttt{false} \rhd C|\!\}$
$\{\!|\texttt{not}\,\texttt{true} \rhd C|\!\} \to \{\!|\texttt{false} \rhd C|\!\}$
$\{\!|\texttt{not}\,\texttt{false} \rhd C|\!\} \to \{\!|\texttt{true} \rhd C|\!\}$

$\{\!|\texttt{if}\,\texttt{true}\,\texttt{then}\, St_1 \,\texttt{else}\, St_2 \rhd C|\!\} \to \{\!|St_1 \rhd C|\!\}$
$\{\!|\texttt{if}\,\texttt{false}\,\texttt{then}\, St_1 \,\texttt{else}\, St_2 \rhd C|\!\} \to \{\!|St_2 \rhd C|\!\}$
$\{\!|\texttt{skip}; St \rhd C|\!\} \to \{\!|St \rhd C|\!\}$
$\{\!|\{St\} \rhd C|\!\} \to \{\!|St \rhd C|\!\}$
$\{\!|X\texttt{:=}I \rhd C|\!\}, \{\!|(X, I') \rhd \sigma|\!\} \to \{\!|\texttt{skip} \rhd C|\!\}, \{\!|(X, I) \rhd \sigma|\!\}$
$\{\!|\texttt{while}\, B\ St \rhd C|\!\} \to \{\!|\texttt{if}\, B \,\texttt{then}\, (St; \texttt{while}\, B\ St) \,\texttt{else}\,\texttt{skip} \rhd C|\!\}$
$\{\!|\texttt{halt}\, I \rhd C|\!\}, \sigma \to I$

$\{\!|\texttt{skip}.I \rhd C|\!\}, \sigma \to I$

Table 12: The *CHAM* language definition

	N nested loops(1..2)				Collatz' conjecture
n	15	16	18	Memory for 18	up to 300
ASF+SDF	1.7	2.9	11.6	13mb	265.1
BC	0.3	0.6	2.3	<1mb	13.8
Haskell	0.3	0.7	2.8	4mb	32.1
Haskell (monads)	0.6	1.4	4.4	3mb	58.7
Maude	3.8	7.7	31.5	6mb	184.5
Ocaml	0.5	1.1	5.0	1mb	10.2
Ocaml (monads)	0.5	0.9	3.8	2mb	21.5
Prolog	1.6	1.9	7.6	316mb	-
Scheme	3.8	7.4	30.2	13mb	122.3

Table 13: Execution times for Big Step definitions

	N nested loops(1..2)				Collatz' conjecture
n	15	16	18	Memory for 18	up to 300
ASF+SDF	11.9	25.7	115.0	9mb	769.6
Haskell	3.2	7.0	31.64	3mb	167.4
Maude	63.4	131.2	597.4	6mb	>1000
Ocaml	1.0	2.2	9.9	1mb	21.0
Prolog	7.0	14.5	-	>700mb	-

Table 14: Execution times for Small Step definitions

	N nested loops(1..2)					Collatz' conjecture
N	9	15	16	18	Memory for 18	up to 300
ASF+SDF	0.6	88.7	214.4	1008.6	10mb	891.3
Haskell	0.1	5.8	12.0	53.9	3mb	157.2
Maude	3.7	552.1	1239	6142.5	6mb	>1h
Ocaml	0.0	1.8	3.8	16.7	1mb	11.0
Prolog	0.1	9.4	-	-	>700mb	-
Scheme (PLT-Redex)	198.2	-	-	-	>700mb	-

Table 15: Execution times for Context Reduction definitions

	N nested loops(1..2)				Collatz' conjecture
N	15	16	18	Memory for 18	up to 300
ASF+SDF	2.5	4.7	18.3	13mb	344.7
Haskell	0.6	1.1	4.4	4mb	41.1
Maude	8.4	15.6	63.2	7mb	483.9
Ocaml	0.5	1.1	5.0	1mb	10.9
Prolog	3.0	6.2	24.0	≈500mb	-
Scheme	5.9	11.3	45.2	10mb	323.6

Table 16: Execution times for Continuation based definitions

limit ourselves to making some high-level remarks. Also, we do not discuss any of the approaches, such as SOS, MSOS, context reduction, or the Cham, which we have already discussed in the body of the paper.

Algebraic denotational semantics. This approach, (see [92, 37, 13, 60] for early papers and [36, 85] for two more recent books), is a special case of RLS, namely, the case in which the rewrite theory $\mathcal{R}_{\mathcal{L}}$ defining language $\mathcal{L}$ is an equational theory. While algebraic semantics shares a number of advantages with RLS, its main limitation is that it is well-suited for giving semantics to *deterministic* languages, but not well-suited for concurrent language definitions. At the model-theoretic level, initial algebra semantics, pioneered by Joseph Goguen, is the preferred approach (see, for example, [37, 36]), but other approaches, based on loose semantics or on final algebras, are also possible.

Other RLS work. RLS is a collective international project. Through the efforts of various researchers, there is by now a substantial body of work demonstrating the usefulness of this approach [11, 87, 83, 81, 51, 86, 20, 70, 88, 31, 29, 41, 12, 53, 55, 17, 16, 30, 23, 72, 3, 82, 25, 73, 43, 40, 34, 4, 28]. A first snapshot of the RLS project was given in [55], and a second in [54]. This paper can be viewed as third snapshot focusing on the variety of definitional styles supported. In particular, a substantial body of experience in giving programming language definitions, and using those definitions both for execution and for analysis purposes has already been gathered. For example, Java 1.4 (see also [19] for a complete formal semantics) and the JVM (see [31, 28]) have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [31, 29]. A semantics of a Caml-like language with threads was discussed in detail in [55], and a modular rewriting logic semantics of a subset of CML has been given in [17] using the Maude MSOS tool [18]. A definition of the Scheme language has been given in [25]. Other language case studies, all specified in Maude, include BC [12], CCS [87, 12], CIAO [82], Creol [41], ELOTOS [86], MSR [15, 80], PLAN [81, 82], the ABEL hardware description language [43], SILF [40], FUN [71], Orc [4], and the π-calculus [83].

Higher-order approaches. The most classic higher-order approach, although not exactly operational, is *denotational semantics* [75, 76, 74, 61]. Denotational semantics has some similarities with its first-order algebraic cousin mentioned above, since both are based on semantic equations. Two differences are: (i) the use of first-order equations in the algebraic case versus the higher-order ones in traditional denotational semantics; and (ii) the kinds of models used in each case. A related class of higher-order approaches uses higher-order functional languages or higher-order theorem provers to give operational semantics to programming languages. Without trying to be comprehensive, we can mention, for example, the use of Scheme in [33], the use of ML in [67], and the use of Common LISP within the ACL2 prover in [44]. There is also a body of work on using monads [59, 91, 45] to implement language interpreters in higher-order functional languages; the monadic approach has better modularity characteristics than standard SOS. A third class of higher-order approaches are based on the use of higher-order abstract syntax (HOAS) [66, 39] and higher-order logical frameworks, such as LF [39] or λ-Prolog [64], to encode programming languages as formal logical systems. For a good example of recent work in this direction see [56] and references there.

Logic-programming-based approaches. Going back to the Centaur project [10, 24], logic programming has been used as a framework for SOS language definitions. Note that λ-Prolog [64] belongs both in this category and in the higher-order one. For a recent textbook giving logic-programming-based language definitions, see [77].

Abstract state machines. Abstract State Machine (ASM) [38] can encode any computation and have a rigorous semantics, so any programming language can be defined as an ASM and thus implicitly be given a semantics. Both big- and small-step ASM semantics have been investigated. The semantics of various programming languages, including, for example, Java [78], has been given using ASMs. There are interesting connections between ASMs and rewriting logic, but their discussion is beyond the scope of this paper.

13 Conclusions

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

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

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. In *Proc. POPL'90*, pages 31–46. ACM, 1990.

[2] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, pages 213–239. ENTCS, Vol. 153, Elsevier, 2006.

[3] W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for java verification against a rewriting semantics. In *Proc. LPAR 2006*, volume 3835 of *LNCS*, pages 412–426. Springer-Verlag, 2005.

[4] M. Al-Turki. A rewriting logic approach to the semantics of Orc. Master's thesis, CS Dept. Univ. of Illinois at Urbana-Champaign, December 2005.

[5] J.-P. Banâtre and D. L. Mètayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[6] Z.-E.-A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. lambda-nu, a calculus of explicit substitutions which preserves strong normalisation. *J. Funct. Program.*, 6(5):699–722, 1996.

[7] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[8] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN *V 3.4 User Manual*. LORIA, Nancy (France), fourth edition, January 2000.

[9] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.

[10] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In *Software Development Environments (SDE)*, pages 14–24, 1988.

[11] C. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics.* PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[12] C. Braga and J. Meseguer. Modular rewriting semantics in practice. In *Proc.* WRLA'04, volume 117, pages 393–416. ENTCS, Elsevier, 2004.

[13] M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Trans. on Prog. Lang. and Systems*, 9(1):54–99, Jan. 1987.

[14] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proceedings of ICALP 2003, 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Springer LNCS*, pages 252–266, 2003.

[15] I. Cervesato and M.-O. Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. In P. Degano, editor, *Proc. Fifth International Workshop on Rewriting Logic and its Applications (WRLA'2004)*, volume 117. Elsevier ENTCS, 2004. Barcelona, Spain, March 27 - 28, 2004.

[16] F. Chalub. An Implementation of Modular SOS in Maude. Master's thesis, Universidade Federal Fluminense, May 2005. `http://www.ic.uff.br/~frosario/dissertation.pdf`.

[17] F. Chalub and C. Braga. A modular rewriting semantics for CML. *J. UCS*, 10(7):789–807, 2004.

[18] F. Chalub and C. Braga. Maude MSOS tool. In G. Denker and C. Talcott, editors, *6th International Workshop on Rewriting Logic and its Applications (WRLA'06)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, to appear.

[19] F. Chen and G. Roşu. Rewriting Logic Semantics of Java 1.4. `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java`.

[20] F. Chen, G. Roşu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Springer LNCS*, pages 197–207, 2003.

[21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[22] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude.* Springer LNCS Vol. 4350, 2007. To appear.

[23] M. Clavel and J. Santa-Cruz. ASIP + ITP: A verification tool based on algebraic semantics. In *Proc. PROLE'05*, 2005. To appear, `http://maude.sip.ucm.es/~clavel/pubs/`.

[24] D. Clément, J. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In K. Fuchi and M. Nivat, editors, *Proceedings, France-Japan AI and CS Symposium*, pages 49–89. ICOT, 1986. Also, Information Processing Society of Japan, Technical Memorandum PL-86-6.

[25] M. d'Amorim and G. Roşu. An Equational Specification for the Scheme Language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005. Selected papers from the 9th Brazilian Symposium on Programming Languages (SBLP'05). Also Technical Report No. UIUCDCS-R-2005-2567, April 2005.

[26] R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[27] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In *Proc. Strategies 2006*.

[28] A. Farzan. *Static and dynamic formal analysis of concurrent systems and languages: a semantics-based approach*. PhD thesis, University of Bergen, Norway, 2007.

[29] A. Farzan, F. Cheng, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. in Proc. CAV'04, Springer LNCS, 2004.

[30] A. Farzan and J. Meseguer. Making partial order reduction tools language-independent. In G. Denker and C. Talcott, editors, *Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2006. To appear.

[31] A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. in Proc. *AMAST'04*, Springer LNCS 3116, 132–147, 2004.

[32] M. Felleisen and D. P. Friedman. Control operators, the secd-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Ebberup, Denmark, Aug. 1986.

[33] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, 2nd edition, 2001.

[34] A. Garrido, J. Meseguer, and R. Johnson. Algebraic semantics of the C preprocessor and correctness of its refactorings. Technical Report UIUCDCS-R-2006-2688, CS Dept., University of Illinois at Urbana-Champaign, February 2006.

[35] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[36] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[37] J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, pages 292–309. Springer-Verlag, 1981. LNCS, Volume 107.

[38] Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press, 1994.

[39] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. 2^{nd} LICS Conf.*, pages 194–204. IEEE, Computer Society Press, 1987.

[40] M. Hills, T. F. Şerbănuţă, and G. Roşu. A rewrite framework for language definitions and for generation of efficient interpreters. In *Proceedings of WRLA'06*. Elsevier, 2006. To appear.

[41] E. B. Johnsen, O. Owe, and E. W. Axelsen. A runtime environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications*, volume 117. ENTCS, Elsevier, 2004.

[42] G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[43] M. Katelman and J. Meseguer. A rewriting semantics for abel with applications to hardware/software co-design and analysis. In G. Denker and C. Talcott, editors, *Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2006. To appear.

[44] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[45] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'95*, pages 333–343. ACM Press, 1995.

[46] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd. Edition*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Tech. Report SRI-CSL-93-05, August 1993.

[47] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

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

[49] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. CONCUR'96, Pisa, August 1996*, pages 331–372. Springer LNCS 1119, 1996.

[50] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.

[51] J. Meseguer. Software specification and verification in rewriting logic. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software, NATO Advanced Study Institute, Marktoberdorf, Germany, July 30 – August 11, 2002*, pages 133–193. IOS Press, 2003.

[52] J. Meseguer. A rewriting logic sampler. In *Proc. International Colloquium on Theoretical Aspects of Computing ICTAC05 (Hanoi, Vietnam, October 2005)*, volume 3722 of *LNCS*, pages 1–28. Springer, 2005.

[53] J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. in Proc. AMAST'04, Springer LNCS 3116, 364–378, 2004.

[54] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2006.

[55] J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.

[56] D. Miller. Representing and reasoning with operational semantics. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 4–20. Springer, 2006.

[57] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[58] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[59] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[60] P. D. Mosses. Unified algebras and action semantics. In *Proc. Symp. on Theoretical Aspects of Computer Science, STACS'89*, pages 17–35. Springer LNCS 349, 1989.

[61] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B, Chapter 11*. North-Holland, 1990.

[62] P. D. Mosses. Pragmatics of modular SOS. In H. Kirchner and C. Ringeissen, editors, *AMAST*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.

[63] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60–61:195–228, 2004.

[64] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth Int. Joint Conf. and Symp. on Logic Programming*, pages 810–827. The MIT Press, 1988.

[65] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. ENTCS, Elsevier, 2004. Proc. 5th Intl. Workshop on Rewriting Logic and its Applications.

[66] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88*, pages 199–208. ACM Press, 1988.

[67] B. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[68] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

[69] J. C. Reynolds. The Discoveries of Continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.

[70] G. Roşu, R. P. Venkatesan, J. Whittle, and L. Leustean. Certifying optimality of state estimation programs. In *Computer Aided Verification (CAV'03)*, pages 301–314. Springer, 2003. LNCS 2725.

[71] G. Roşu. K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2005-2672, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[72] R. Sasse. Taclets vs. rewriting logic – relating semantics of Java. Master's thesis, Fakultät für Informatik, Universität Karlsruhe, Germany, May 2005. Technical Report in Computing Science No. 2005-16, `http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2005/16`.

[73] R. Sasse and J. Meseguer. Java+itp: A verification tool based on hoare logic and algebraic semantics. In G. Denker and C. Talcott, editors, *Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications.* ENTCS, Elsevier, 2006. To appear.

[74] D. A. Schmidt. *Denotational Semantics – A Methodology for Language Development.* Allyn and Bacon, Boston, MA, 1986.

[75] D. Scott. Outline of a mathematical theory of computation. In *Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176. Princeton University, 1970. Also appeared as Technical Monograph PRG 2, Oxford University, Programming Research Group.

[76] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Microwave Research Institute Symposia Series, Vol. 21: Proc. Symp. on Computers and Automata.* Polytechnical Institute of Brooklyn, 1971.

[77] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages.* Addison-Wesley, 1995.

[78] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.

[79] M.-O. Stehr. CINNI - a generic calculus of explicit substitutions and its application to lambda-, sigma- and pi-calculi. ENTCS 36, Elsevier, 2000. Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.

[80] M.-O. Stehr, I. Cervesato, and S. Reich. An execution environment for the MSR cryptoprotocol specification language. `http://formal.cs.uiuc.edu/stehr/msr.html`.

[81] M.-O. Stehr and C. Talcott. PLAN in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*, volume 117. ENTCS, Elsevier, 2002.

[82] M.-O. Stehr and C. L. Talcott. Practical techniques for language design and prototyping. In J. L. Fiadeiro, U. Montanari, and M. Wirsing, editors, *Abstracts Collection of the Dagstuhl Seminar 05081 on Foundations of Global Computing. February 20 – 25, 2005. Schloss Dagstuhl, Wadern, Germany.*, 2005.

[83] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[84] M. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[85] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.

[86] A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.

[87] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[88] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in maude. *J. Log. Algebr. Program.*, 67(1-2):226–293, 2006.

[89] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.

[90] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003.

[91] P. Wadler. The essence of functional programming. In *POPL*, pages 1–14, 1992.

[92] M. Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980.

[93] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.