Information and Computation, Volume 207(2), pp 305-340. 2009

A Rewriting Logic Approach to Operational Semantics ⋆

Traian Florin Șerbănuță [a,b] Grigore Roșu [a] José Meseguer [a]

[a] *Department of Computer Science, University of Illinois at Urbana-Champaign.*

[b] *Faculty of Mathematics and Informatics, University of Bucharest, Romania*

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.

Key words: operational semantics, rewriting logic, rewriting logic semantics

1 Introduction

This paper is part of the rewriting logic semantics (RLS) project (see [56, 57] and the references there). The broad goal of the project is to develop a tool-

⋆ Supported by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, NASA grant NNL08AA23C, and by several Microsoft gifts. This paper is a full version (including detailed proofs, more detailed explanations and constructions, and some further results) of the SOS'07 workshop paper [T-F. Șerbănuță, G. Roșu, J. Meseguer, A Rewriting Logic Approach to Operational Semantics (Extended Abstract), Electronic Notes in Theoretical Computer Science 192(1), Elsevier, 2007, pp. 125–141]

Email addresses: tserban2@cs.uiuc.edu (Traian Florin Șerbănuță), grosu@cs.uiuc.edu (Grigore Roșu), meseguer@cs.uiuc.edu (José Meseguer).

supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on *rewriting logic* [50].

Any logical framework worth its salt should be evaluated in terms of its expressiveness and flexibility. Regarding expressiveness, a very pertinent question is: how does RLS express various approaches to operational semantics? In particular, how well can it express various approaches in the SOS tradition? The goal of this paper is to provide an answer to these questions. Partial answers, giving detailed comparisons with specific approaches have appeared elsewhere. For example, [48] and [91] provide comparisons with standard SOS [70]; [55] compares RLS with both standard SOS and Mosses' modular structural operational semantics (MSOS) [65]; and [50] compares RLS with chemical abstract machine (Cham) semantics [8]. However, no comprehensive comparison encompassing most approaches in the SOS tradition has been given to date. To make our ideas more concrete, in this paper we use a simple programming language, show how it is expressed in each different definitional style, and how that style can be faithfully captured as a rewrite theory in the RLS framework. We furthermore prove correctness theorems showing the faithfulness of the RLS representation for each style. Even though we exemplify the techniques and proofs with a simple language for concreteness' sake, the process of representing each definitional style in RLS and proving the faithfulness of the representation is completely general and mechanical, and in some cases like MSOS has already been automated [19]. The range of styles covered includes: big-step (or natural) SOS semantics; small-step SOS semantics; MSOS semantics; context-sensitive reduction semantics; continuation-based semantics; and Cham semantics.

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

1.1 Challenges

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

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

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

(3) **Q**: *Can you handle higher-order syntax?*
A: Rewriting logic, cannot *directly* handle higher-order syntax with bindings and reasoning modulo α-conversion. However, it is well-known that higher-order syntax admits first-order representations, such as explicit substitution calculi and de Bruijn numbers, e.g., [1, 7, 82]. However, the granularity of computations is changed in these representations; for example, a single 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 [82], 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 [34, 71] are traditionally understood as higher-order functions. Using the above-mentioned explicit calculi they can be represented in a first-order way. In Section 9 we present an alternative view of continuations that is intrinsically first-order in the style of, e.g., Wand [95], and prove a theorem showing that, for our language, first-order continuation semantics and context-sensitive reduction semantics are equivalent as rewrite theories in RLS. We also emphasize that in a computational logical framework, continuations are not just a means of implementing a language, but can be used to actually *define* the semantics of a language.

(5) **Q**: *Can you handle concurrency?*

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

(6) **Q**: *How expressive is the framework?*

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

(7) **Q**: *Is anything lost in translation?*

A: This is a very important question, because the worth of a logical framework does not just depend on whether something can be represented "in principle," but on *how well* it is represented. The key point is to have a very small *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 [53].

(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 mechanically generated interpreters from RLS definitions for our example language using various systems for the different styles considered. Generally speaking, these performance figures are very encouraging and show that good performance interpreters can be directly obtained from RLS semantic definitions. Although for this paper we have used hand-made (though mechanical) translations to the presented implementation languages, we envision a toolkit which would use RLS as a common framework for different definitional styles having as back-ends multiple execution languages.

1.2 Benefits

Our skeptical language designer could still say,

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

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

Besides the benefits already mentioned in our answers to questions in Section 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 an RLS operational semantics definition and an implemen-

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

A fourth benefit comes from the availability in RLS of what we call the "abstraction dial," which can be used to reach a good balance between abstraction and computational observability in semantic definitions. The point is which *computational granularity* is appropriate. A small-step semantics opts for very fine-grained computations. But this is not necessarily the only or the best option for all purposes. The fact that an RLS theory's axioms include both equations and rewrite rules provides the useful "abstraction dial," because rewriting takes place *modulo* the equations. That is, computations performed by equations are abstracted out and become *invisible*. This has many advantages, as explained in [57]. For example, for formal analysis it can provide a huge reduction in search space for model checking purposes, which is one of the reasons why the Java model checkers described in [31, 33] perform so well.

For language definition purposes, this again has many advantages. For example, in Sections 6 and 5, we use equations to define the semantic infrastructure (stores, etc.) of SOS definitions; in Section 8 equations are also used to hide the extraction and application of evaluation contexts, which are "meta-level" operations, carrying no computational meaning; in Section 9, equations are also used to decompose the evaluation tasks into their corresponding subtasks; finally, in Sections 7 and 10, equations of associativity and commutativity are used to achieve, respectively, modularity of language definitions, and true concurrency in chemical-soup-like computations. The point in all these cases is always the same: to achieve the right granularity of computations.

1.3 Outline of the paper

The remainder of this paper is organized as follows. Section 2 presents basic concepts of rewriting logic and recalls its deduction rules and its relationship with equational logic and term rewriting. Section 3 introduces a simple imperative language that will be used in the rest of the paper to discuss the various definitional styles and their RLS representations. Section 4 gathers 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 [50] is a computational logic that can be efficiently implemented and that has good properties as a general and flexible *logical and semantic framework*, in which a wide range of logics and models of computation can be faithfully represented [48]. In particular, for programming language semantics it provides the RLS framework, of which we emphasize the operational semantics aspects in this paper (for the mathematical aspects of RLS see [56, 57]).

Two key points to explain are: (i) how rewriting logic combines equational logic and traditional term rewriting; and (ii) what the intuitive meaning of a rewrite theory is all about. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ 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 *non-equational* semantics. This is the whole *raison d'être* 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 rewriting 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* [52] having sorts (denoted s, s', s'', etc.), subsort inclusions, and kinds (denoted k, k', k'', etc.), which gather together connected components of sorts. Kinds allow error terms like $3/0$, which has a kind but no sort. Similar inference rules can be given for untyped or simply typed

(many-sorted) versions of the logic. Given $\mathcal{R} = (\Sigma, E, R)$, 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 $f : s_1 \dots s_n \longrightarrow s$ in Σ, with $t_i \in T_\Sigma(X)_{s_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{s_{j_l}}$, $1 \leq l \leq m$,

$$\frac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad \dots \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, \dots, t_{j_1}, \dots, t_{j_m}, \dots, t_n) \longrightarrow f(t_1, \dots, t'_{j_1}, \dots, t'_{j_m}, \dots, t_n)}$$

- **Replacement**. For each $\theta : X \longrightarrow T_\Sigma(Y)$ and for each rule in R of the form

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

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

  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.

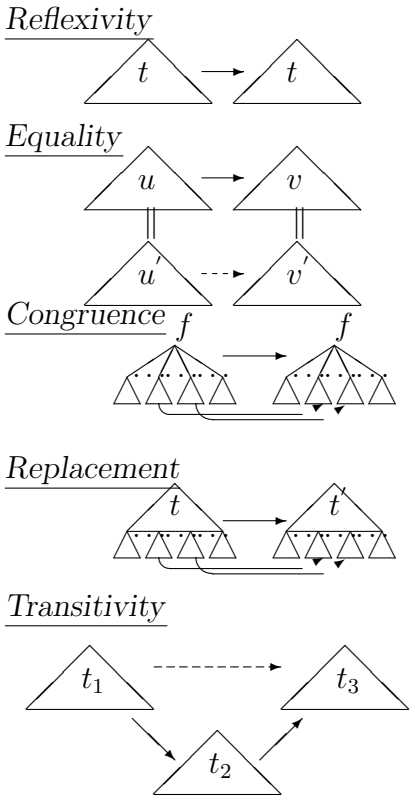

Fig. 1. Visual representation of rewriting logic deduction.

A somewhat more general version of rewriting logic [15] 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 non-frozen 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 form $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$, makes possible a more expressive control of the possibility of rewriting under contexts already supported by the **Congruence** rule; that is, it endows rewrite theories with flexible context-sensitive rewriting capabilities. [1]

Note that, in general, a proof $\mathcal{R} \vdash t \longrightarrow t'$ 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

[1] We will not consider this general version. The interested reader is referred to [15].

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. However, it should be noted that rewriting logic *does not* have a builtin "one-step" rewrite relation, that being the reason for which we need a methodology to encode "one step"-based formalisms such as SOS semantics. The "one-step" relation we define above is only at the deduction level and is introduced solely to help our proofs.

The whole point of RLS is then to define the semantics of a programming language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$. RLS uses the fact that rewriting logic deduction is performed *modulo* the equations in $\mathcal{R}_{\mathcal{L}}$ to faithfully capture the desired granularity of a language's computations. This is achieved by making rewriting rules all intended computational steps, while using equations for convenient equivalent structural transformations of the state, or auxiliary "infrastructure" computations, which should not be regarded as computation steps. Note that this does not preclude performing also equational simplification with equations. That is, the set E of equations in a rewrite theory can often be fruitfully decomposed as a disjoint union $E = E_0 \cup A$, where A is a set of *structural axioms*, such as associativity, commutativity and identity of some function symbols, and E_0 is a set of equations that are confluent and terminating *modulo* the axioms A. A rewrite engine supporting rewriting modulo A will then execute both the equations E_0 and the rules R modulo A by rewriting. Under a condition called *coherence* [92], this form of execution then provides a complete inference system for the given rewrite theory (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 with R are E-equivalence classes, so that the E_0-steps become *invisible*. Operationally, the execution of rules R and equations E_0 must be kept separate for soundness reasons. This is particularly apparent in the case of executing *conditional* equations and rules: for a conditional equation it would be *unsound* to use rules in R to evaluate its condition; and for a conditional rule it would likewise be unsound to use rules in R to evaluate the *equational* part of its condition.

There are many systems that either specifically implement term rewriting efficiently, so-called as *rewrite engines*, or support term rewriting as part of a more complex functionality. Any of these systems can be used as an underlying platform for execution and analysis of programming languages defined using the techniques proposed in this paper. Without attempting to be exhaustive, we here only mention (alphabetically) some engines that we are more familiar with, noting that many functional languages and theorem provers provide support for term rewriting as well: ASF+SDF [87], CafeOBJ [28], Elan [9],

$$
\begin{array}{rl}
AExp ::= & Var \mid Int \mid AExp \texttt{ + } AExp \mid AExp \texttt{ - } AExp \mid AExp \texttt{ * } AExp \mid \\
& AExp \texttt{ / } AExp \mid \texttt{ ++ } Var \\
BExp ::= & Bool \mid AExp \texttt{ <= } AExp \mid AExp \texttt{ >= } AExp \mid AExp \texttt{ == } AExp \mid \\
& BExp \texttt{ and } BExp \mid BExp \texttt{ or } BExp \mid \texttt{not } BExp \\
Stmt ::= & \texttt{skip} \mid Var \texttt{ := } AExp \mid Stmt \texttt{ ; } Stmt \mid \texttt{\{ } Stmt \texttt{\} } \mid \\
& \texttt{if } BExp \texttt{ then } Stmt \texttt{ else } Stmt \mid \texttt{while } BExp \ Stmt \mid \texttt{halt } AExp \\
Pgm ::= & Stmt \texttt{ . } AExp
\end{array}
$$

Table 1
A Small Imperative Language

Maude [22], OBJ [38], and Stratego [93]. Some of these engines can achieve remarkable speeds on today's machines, in the order of tens of millions of rewrite steps per second.

3 A Simple Imperative Language

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

The semantics of $\texttt{++}x$ 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 remainder of the paper: $X \in Var$, $A \in AExp$, $B \in BExp$, $St \in Stmt$, $P \in Pgm$, $I \in Int$, $T \in Bool = \{true, false\}$, 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 [8, 44, 65, 70, 96] for further details on the described operational semantics styles.

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

4 Store

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

To define the store, we assume a pairing "binding" constructor "$\_ \mapsto \_$", associating values to variables[2], and an associative and commutative union operation "$\_\ \_$" 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 $\_\ \_$:

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

$$(S\ X \mapsto I)[X] = I$$
$$(S\ X \mapsto I)[X'] = S[X'] \,\mathbf{if}\, X \neq X'$$
$$(S\ X \mapsto I)[X \leftarrow I'] = S\ X \mapsto I'$$
$$(S\ X \mapsto I)[X' \leftarrow I'] = S[X' \leftarrow I']\ X \mapsto I \,\mathbf{if}\, X \neq X'$$
$$\emptyset[X \leftarrow I] = X \mapsto I$$

Note the $X \neq X$ appearing as a condition is not a negative condition, but rather a Boolean predicate, which can be equationally defined for any constructor-based type such as the type of variables, for example. Since these definitions are equational, from a rewriting logic semantic point of view they are invisible: transitions are performed *modulo* these equations. This way we can maintain a coarser computational granularity, while making use of auxiliary functions defined using equations. Although it might seem that, by using built-ins as integers and names, one cannot guarantee the existence of the initial model, notice that all the "built-ins" appearing in these definitions (names, booleans, integers) are definable as initial models of corresponding equational theories. And indeed, when performing formal proofs, one will make use of these equational definitions of the so-called built-ins. A store s is *well-formed* if $\mathcal{E}_{Store} \vdash s = x_1 \mapsto i_1 \ldots x_n \mapsto i_n$ for some $x_j \in Var$ and $i_j \in Int$, 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 $\sigma : Var \xrightarrow{\circ} Int$, written $s \simeq \sigma$, if s is well-formed and behaves as σ, that is, if for any $x \in Var, i \in Int$, $\sigma(x) = i$ 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 $x, x' \in Var$, $i, i' \in Int$, $s, s' \in Store$ and finite partial functions $\sigma, \sigma' : Var \xrightarrow{\circ} Int$.*

(1) $\emptyset \simeq \perp$ where $\perp$ is the function undefined everywhere.
(2) $(s\ x \mapsto i) \simeq \sigma$ implies that $s \simeq \sigma[\perp\ /x]$ where $\sigma[\perp\ /x]$ is defined as σ restricted to $Dom(\sigma) \setminus \{x\}$.
(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 $x \in Var, i \in Int$.
(2) Let σ' be such that $s \simeq \sigma'$. We will prove that $Dom(\sigma') = Dom(\sigma) \setminus \{x\}$ and for any $x' \in Dom(\sigma')$, $\sigma'(x) = \sigma(x)$. 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-formedness of $s\ x \mapsto i$; therefore, σ' is not defined on x'. If $x' \neq x$, then $\mathcal{E}_{Store} \vdash s[x'] = (s\ x \mapsto i)[x']$, therefore σ' is defined on x' iff σ is defined on x', and if so

$\sigma'(x') = \sigma(x')$.

(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 $\mathcal{E}_{Store} \nvdash s[x] = i$ for any i, whence $\mathcal{E}_{Store} \nvdash s = s'\ x \mapsto i$. 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 $\mathcal{E}_{Store} \nvdash s = \emptyset$, 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 $\mathcal{E}_{Store} \vdash s[x \leftarrow i] = \cdots = (x_1 \mapsto i_1 \ldots x_i \mapsto i_i)[x \leftarrow i](x_{i+1} \mapsto i_{i+1} \ldots x_n \mapsto i_n) = \cdots = \emptyset[x \leftarrow i]s = (x \mapsto i)s = s(x \mapsto i)$.

□

In the following, we will use symbols S, S', S_1, ..., to denote variables of type *Store*.

5 Big-Step Operational Semantics

Introduced as natural semantics in [44], also named relational semantics in [60], or evaluation semantics, big-step semantics is "the most denotational" of the operational semantics. One can view big-step definitions as definitions of functions interpreting each language construct in an appropriate domain.

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

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

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 big-step operational semantics definition for our simple language, except its `halt` statement (which is discussed at the end of this section), which we call *BigStep*, is presented in Table 2. We choose to exclude

Types of configurations: $\langle Int, Store\rangle$, $\langle Bool, Store\rangle$, $\langle AExp, Store\rangle$, $\langle BExp, Store\rangle$, $\langle Stmt, Store\rangle$, $\langle Pgm\rangle$, $\langle Int\rangle$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 2
The *BigStep* language definition

$$\langle X, S\rangle \to \langle S[X], S\rangle$$

$$\langle \texttt{++}X, S\rangle \to \langle I, S[X \leftarrow I]\rangle \ \textbf{if}\ I = S[X] + 1$$

$$\langle A_1 + A_2, S\rangle \to \langle I_1 +_{Int} I_2, S_2\rangle \ \textbf{if}\ \langle A_1, S\rangle \to \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \to \langle I_2, S_2\rangle$$

$$\langle A_1 \texttt{<=} A_2, S\rangle \to \langle (I_1 \leq_{Int} I_2), S_2\rangle \ \textbf{if}\ \langle A_1, S\rangle \to \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \to \langle I_2, S_2\rangle$$

$$\langle B_1 \ \texttt{and}\ B_2, S\rangle \to \langle T, S_2\rangle \ \textbf{if}\ \langle B_1, S\rangle \to \langle \mathit{true}, S_1\rangle \wedge \langle B_2, S_1\rangle \to \langle T, S_2\rangle$$

$$\langle B_1 \ \texttt{and}\ B_2, S\rangle \to \langle \mathit{false}, S_1\rangle \ \textbf{if}\ \langle B_1, S\rangle \to \langle \mathit{false}, S_1\rangle$$

$$\langle \texttt{not}\ B, S\rangle \to \langle \mathit{not}(T), S'\rangle \ \textbf{if}\ \langle B, S\rangle \to \langle T, S'\rangle$$

$$\langle \mathit{skip}, S\rangle \to \langle S\rangle$$

$$\langle X \texttt{:=} A, S\rangle \to \langle S'[X \leftarrow I]\rangle \ \textbf{if}\ \langle A, S\rangle \to \langle I, S'\rangle$$

$$\langle St_1; St_2, S\rangle \to \langle S'\rangle \ \textbf{if}\ \langle St_1, S\rangle \to \langle S''\rangle \wedge \langle St_2, S''\rangle \to \langle S'\rangle$$

$$\langle \{St\}, S\rangle \to \langle S'\rangle \ \textbf{if}\ \langle St, S\rangle \to \langle S'\rangle$$

$$\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2, S\rangle \to \langle S_2\rangle \ \textbf{if}\ \langle B, S\rangle \to \langle \mathit{true}, S_1\rangle \wedge \langle St_1, S_1\rangle \to \langle S_2\rangle$$

$$\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2, S\rangle \to \langle S_2\rangle \ \textbf{if}\ \langle B, S\rangle \to \langle \mathit{false}, S_1\rangle \wedge \langle St_2, S_1\rangle \to \langle S_2\rangle$$

$$\langle \texttt{while}\ B\ St, S\rangle \to \langle S'\rangle \ \textbf{if}\ \langle B, S\rangle \to \langle \mathit{false}, S'\rangle$$

$$\langle \texttt{while}\ B\ St, S\rangle \to \langle S'\rangle \ \textbf{if}\ \langle B, S\rangle \to \langle \mathit{true}, S_1\rangle \wedge \langle St, S_1\rangle \to \langle S_2\rangle \wedge \langle \texttt{while}\ B\ St, S_2\rangle \to \langle S'\rangle$$

$$\langle St.A\rangle \to \langle I\rangle \ \textbf{if}\ \langle St, \emptyset\rangle \to \langle S\rangle \wedge \langle A, S\rangle \to \langle I, S'\rangle$$

Table 3
$\mathcal{R}_{BigStep}$ rewriting logic theory

from the presentation the semantics for constructs entirely similar to the ones presented, such as "$-$", "$*$", "$/$" and "`or`". 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 3. Note that, because the rewriting relation is reflexive, we did not need to add the reflexivity rules for boolean and integer values.

Due to the one-to-one correspondence between big-step rules in *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) $\mathcal{R}_{BigStep} \vdash \langle p\rangle \to^1 \langle i\rangle$

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 $\sigma : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ 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 $a \in \mathit{AExp}, i \in \mathit{Int}, \sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ 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 $b \in \mathit{AExp}, t \in \mathit{Bool}, \sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ 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 $st \in \mathit{Stmt}, \sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ 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 $\mathcal{R}_{BigStep} \vdash \langle b, s\rangle \rightarrow^1 \langle true, s_1\rangle$, $s_1 \simeq \sigma_1$

and $\mathcal{R}_{BigStep} \vdash \langle st, s_1\rangle \rightarrow^1 \langle s_2\rangle$, $s_2 \simeq \sigma_2$

and $\mathcal{R}_{BigStep} \vdash \langle \texttt{while}\ b\ st, s_2\rangle \rightarrow^1 \langle s'\rangle$ and $s' \simeq \sigma'$) iff

$\mathcal{R}_{BigStep} \vdash \langle \texttt{while}\ b\ st, s\rangle \rightarrow^1 \langle s'\rangle$ and $s' \simeq \sigma'$.

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

This completes the proof. □

The only apparent difference between *BigStep* and $\mathcal{R}_{BigStep}$ 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, $\mathcal{R}_{BigStep}$ 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. It can be easily and efficiently interpreted in any recursive, functional or logical framework. It is particularly useful for defining type systems.

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

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

6 Small-Step Operational Semantics

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

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

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

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

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

A similar technique is proposed in [55], but there two different types of configurations are employed, one standard and the other "tagged" with the modifier. However, allowing "$\cdot$" 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), which we call *SmallStep*, is presented in Table 4. The corresponding small-step rewriting logic theory $\mathcal{R}_{\mathit{SmallStep}}$ is given in Table 5. The language described here does not involve labels on rules like in the SOS of concurrent systems. For that, one would take an approach similar to that presented in Section 7, that is, pushing the labels back into the configurations.

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 con-

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

Types of configurations: $\langle AExp, Store\rangle$, $\langle BExp, Store\rangle$, $\langle Stmt, Store\rangle$, $\langle Pgm, Store\rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 4
The *SmallStep* language definition

$$\cdot\langle X, S\rangle \to \langle (S[X]), S\rangle$$

$$\cdot\langle \texttt{++}X, S\rangle \to \langle I, S[X \leftarrow I]\rangle \ \textbf{if}\ I = S[X] + 1$$

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

$$\cdot\langle I_1 + A_2, S\rangle \to \langle I_1 + A_2', S'\rangle \ \textbf{if}\ \cdot\langle A_2, S\rangle \to \langle A_2', S'\rangle$$

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

$$\cdot\langle A_1 <= A_2, S\rangle \to \langle A_1' <= A_2, S'\rangle \ \textbf{if}\ \cdot\langle A_1, S\rangle \to \langle A_1', S'\rangle$$

$$\cdot\langle I_1 <= A_2, S\rangle \to \langle I_1 <= A_2', S'\rangle \ \textbf{if}\ \cdot\langle A_2, S\rangle \to \langle A_2', S'\rangle$$

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

$$\cdot\langle B_1 \ \texttt{and}\ B_2, S\rangle \to \langle B_1' \ \texttt{and}\ B_2, S'\rangle \ \textbf{if}\ \cdot\langle B_1, S\rangle \to \langle B_1', S'\rangle$$

$$\cdot\langle \texttt{true and}\ B_2, S\rangle \to \langle B_2, S\rangle$$

$$\cdot\langle \texttt{false and}\ B_2, S\rangle \to \langle \texttt{false}, S\rangle$$

$$\cdot\langle \texttt{not}\ B, S\rangle \to \langle \texttt{not}\ B', S'\rangle \ \textbf{if}\ \cdot\langle B, S\rangle \to \langle B', S'\rangle$$

$$\cdot\langle \texttt{not true}, S\rangle \to \langle \texttt{false}, S\rangle$$

$$\cdot\langle \texttt{not false}, S\rangle \to \langle \texttt{true}, S\rangle$$

$$\cdot\langle X := A, S\rangle \to \langle X := A', S'\rangle \ \textbf{if}\ \cdot\langle A, S\rangle \to \langle A', S'\rangle$$

$$\cdot\langle X := I, S\rangle \to \langle \texttt{skip}, S[X \leftarrow I]\rangle$$

$$\cdot\langle St_1; St_2, S\rangle \to \langle St_1'; St_2, S'\rangle \ \textbf{if}\ \cdot\langle St_1, S\rangle \to \langle St_1', S'\rangle$$

$$\cdot\langle \texttt{skip}; St_2, S\rangle \to \langle St_2, S\rangle$$

$$\cdot\langle \{St\}, S\rangle \to \langle St, S\rangle$$

$$\cdot\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2, S\rangle \to \langle \texttt{if}\ B'\ \texttt{then}\ St_1\ \texttt{else}\ St_2, S'\rangle \ \textbf{if}\ \cdot\langle B, S\rangle \to \langle B', S'\rangle$$

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

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

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

$$\cdot\langle St.A, S\rangle \to \langle St'.A, S'\rangle \ \textbf{if}\ \cdot\langle St, S\rangle \to \langle St', S'\rangle$$

$$\cdot\langle \texttt{skip}.A, S\rangle \to \langle \texttt{skip}.A', S'\rangle \ \textbf{if}\ \cdot\langle A, S\rangle \to \langle A', S'\rangle$$

$$eval(P) = smallstep(\langle P, \emptyset\rangle)$$

$$smallstep(\langle P, S\rangle) = smallstep(\cdot\langle P, S\rangle)$$

$$smallstep(\cdot\langle \texttt{skip}.I, S\rangle) \to I$$

Table 5
$\mathcal{R}_{SmallStep}$ rewriting logic theory

figurations. However, in *SmallStep* it is not the case that all right hand sides are normal forms (this actually is a key feature of small-step semantics). The "$\cdot$" 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 $p \in Pgm$, $\sigma, \sigma' : Var \xrightarrow{\circ} Int$ 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* $\sigma : Var \xrightarrow{\circ} Int$, *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 $s \in Store, \sigma : Var \xrightarrow{\circ} Int$, $s \simeq \sigma$):

(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 $a, a' \in AExp$, $\sigma' : Var \xrightarrow{\circ} Int$ 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 $b, b' \in BExp$, $\sigma' : Var \xrightarrow{\circ} Int$ 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 $st, st' \in Stmt$, $\sigma' : Var \xrightarrow{\circ} Int$ 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 $p, p' \in Pgm$, $\sigma' : Var \xrightarrow{\circ} Int$ 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) $\mathit{SmallStep} \vdash \langle st_1; st_2, \sigma\rangle \to \langle st_1'; st_2, \sigma'\rangle$ iff
$\mathit{SmallStep} \vdash \langle st_1, \sigma\rangle \to \langle st_1', \sigma'\rangle$ iff
$R_{\mathit{SmallStep}} \vdash \cdot\langle st_1, s\rangle \to^1 \langle st_1', s'\rangle$ and $s' \simeq \sigma'$ iff
$R_{\mathit{SmallStep}} \vdash \cdot\langle st_1; st_2, s\rangle \to^1 \langle st_1' + st_2, s'\rangle$ and $s' \simeq \sigma'$.

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

Let us now move to the second equivalence. For this proof let $\to^n$ be the restriction of $\mathcal{R}_{\mathit{SmallStep}}$ relation $\to$ 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 $p \in \mathit{Pgm}$, $\sigma : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ and $s \in \mathit{Store}$ such that $s \simeq \sigma$):

$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to^n \langle p', \sigma'\rangle$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) \to^n \mathit{smallstep}(\cdot\langle p', s'\rangle)$ 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
$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to^n \langle p', \sigma'\rangle$ iff
$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to \langle p_1, \sigma_1\rangle$ and $\mathit{SmallStep} \vdash \langle p_1, \sigma_1\rangle \to^{n-1} \langle p', \sigma'\rangle$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \cdot\langle p, s\rangle \to \langle p_1, s_1\rangle$ and $s_1 \simeq \sigma_1$ (by 1)
and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p_1, s_1\rangle) \to^{n-1} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$ iff
(by the induction hypothesis)
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle p, s\rangle) \to^1 \mathit{smallstep}(\langle p_1, s_1\rangle)$ and $s_1 \simeq \sigma_1$
and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p_1, s_1\rangle) \to^{n-1} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle p, s\rangle) \to^n \mathit{smallstep}(\cdot\langle p', s'\rangle)$ 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, $\mathit{SmallStep} \vdash \langle p, \bot\rangle \to^* \langle \texttt{skip}.i, \sigma\rangle$ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, \emptyset\rangle) \to \mathit{smallstep}(\cdot\langle \texttt{skip}.i, s\rangle)$, $s \simeq \sigma$; the rest follows from $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{eval}(p) = \mathit{smallstep}(\langle p, \emptyset\rangle)$ and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle \texttt{skip}.i, s\rangle) = i$. □

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

Weaknesses. Each small step does the same amount of computation as a big step in finding the next redex. It does not give a "true concurrency" semantics, that is, one has to choose a certain interleaving (no two rules can be applied on the same term at the same time), mainly because reduction is forced

to occur only at the top. One of the reasons for introducing SOS was 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:

$$\cdot\langle \texttt{halt}\ \ A, S\rangle \to \langle \texttt{halt}\ \ A', S'\rangle \quad \textbf{if} \quad \cdot\langle A, S\rangle \to \langle A', S'\rangle$$
$$\cdot\langle \texttt{halt}\ \ I; St, S\rangle \to \langle \texttt{halt}\ \ I, S\rangle$$
$$\cdot\langle \texttt{halt}\ \ I.A, S\rangle \to \langle \texttt{skip}.I, S\rangle$$

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 computational granularity of halt: it should just terminate the execution in *one step*!

7 MSOS Semantics

MSOS semantics was introduced by Mosses in [64, 65] 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 labels on transitions (as provided by SOS), plus a discipline of only selecting needed attributes from the states.

A transition in *MSOS* is of the form $P \xrightarrow{\mathcal{X}} P'$, where P and P' are program expressions and $\mathcal{X}$ is a label describing the structure of the state both before and after the transition. If $\mathcal{X}$ is missing, then the state is assumed to stay unchanged. Specifically, $\mathcal{X}$ is a record containing fields denoting the semantic components; the preferred notation in *MSOS* for saying that in the label $\mathcal{X}$ the semantic component associated to the field name σ (e.g., a store name) is σ_0 (e.g., a function associating values to variables) is $\mathcal{X} = \{\sigma = \sigma_0, \ldots\}$. 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. If record comprehension is used in both the premise and the conclusion of an MSOS rule, then all occurrences of "..." stand for the same fields with the same semantic components. 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, as illustrated by the *MSOS* rule' for assignment:

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

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

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

where "()" stand for monoid unit.

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

The *MSOS* description of the small-step SOS definition in Table 4 is given in Table 6 (we let $\mathcal{X}$ range over labels on transitions).

Because the part of the state not involved in a certain rule is hidden through the "..." notation, language extensions can be made modularly. Consider, for example, adding `halt` to the definition in Table 6. One possible way to do it is to follow the technique proposed in [65] for adding non-parametric abrupt termination, with some modifications to suit our needs to abruptly terminate

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

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

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

$$\frac{I=I_1+_{Int}I_2}{I_1+I_2 \to I}$$

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

$$\frac{T=I_1 \leq_{Int} I_2}{I_1\texttt{<=}I_2 \to T}$$

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

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

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

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

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

$$\{St\} \to St$$

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

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

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

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

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

Table 6
The *MSOS* language definition

the program with a value. For this, we add a write-only field in the record, say *halt?* having as arrows the monoid freely generated by integer numbers, along with a language construct `stuck` to block the execution of the program. To "catch the halt signal" we extend the abstract syntax with a new construct, say **program**, applied to a top-level program. The first set of *MSOS* rules for `halt` are then:

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

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

$$\frac{P \xrightarrow{\{halt?'=I,\ldots\}} P'}{\texttt{program}\ P \xrightarrow{\{halt?'=I,\ldots\}} \texttt{program skip}.I}$$

$$\frac{P \xrightarrow{\{halt?'=(),\ldots\}} P'}{\texttt{program}\ P \xrightarrow{\{halt?'=(),\ldots\}} \texttt{program}\ P'}$$

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

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

$$\frac{\text{unobs}\{halt? = nil, halt?' = nil, \ldots\}}{\texttt{halt}\ I \xrightarrow{\{halt?=nil,halt?'=I,\ldots\}} \texttt{stuck}}$$

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

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

To give a faithful representation of *MSOS* definitions in rewriting logic, we here follow the methodology in [55]. Using the fact that labels describe changes from their source state to their destination state, one can move the labels back into the configurations. That is, a transition step $P \xrightarrow{u} P'$ is modeled as a rewrite step $\cdot\langle P, u^{pre}\rangle \rightarrow \langle P', u^{post}\rangle$, 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:

- For unobservable transitions, $u^{pre} = u^{post}$; same applies for unobservable attributes in premises;
- *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 6 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* [55] and is made precise by the following theorem, strongly relating MSOS and modular rewriting semantics.

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

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

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

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

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

8 Reduction Semantics with Evaluation Contexts

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

(1) it gives a more compact semantics to context-sensitive reduction, by using parsing to find the next redex rather than small-step rules; and
(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, $C[\texttt{halt}\ I] \rightarrow I$, preserving the desired computational granularity. Additionally, one can also incorporate the configuration as part of the evaluation context, and thus have full access to semantic information on a "by need basis"; the PLT-Redex implementation of context reduction, for example, supports this approach. Notice how the assignment rule, for example, modifies both the redex, transforming it to `skip`, and the evaluation context, altering the state which can be found at its top. In this framework, constructs like `call/cc` can be defined with little effort.

$$\cdot\langle X, \{\sigma : S, W\}\rangle \to \langle I, \{\sigma : S, W\}\rangle \ \textbf{if}\ I = S[X]$$

$$\cdot\langle \texttt{++}X, \{\sigma : S_0, W\}\rangle \to \langle I, \{S_0[X \leftarrow I], W\}\rangle \ \textbf{if}\ I = S_0[X] + 1$$

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

$$\cdot\langle I_1 + A_2, R\rangle \to \langle I_1 + A_2', R'\rangle \ \textbf{if}\ \cdot\langle A_2, R\rangle \to \langle A_2', R'\rangle$$

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

$$\cdot\langle A_1 \texttt{<=} A_2, R\rangle \to \langle A_1' \texttt{<=} A_2, R'\rangle \ \textbf{if}\ \cdot\langle A_1, R\rangle \to \langle A_1', R'\rangle$$

$$\cdot\langle I_1 \texttt{<=} A_2, R\rangle \to \langle I_1 \texttt{<=} A_2', R'\rangle \ \textbf{if}\ \cdot\langle A_2, R\rangle \to \langle A_2', R'\rangle$$

$$\cdot\langle I_1 \texttt{<=} I_2, R\rangle \to \langle I_1 \leq_{Int} I_2, R\rangle$$

$$\cdot\langle B_1 \ \texttt{and}\ B_2, R\rangle \to \langle B_1' \ \texttt{and}\ B_2, R'\rangle \ \textbf{if}\ \cdot\langle B_1, R\rangle \to \langle B_1', R'\rangle$$

$$\cdot\langle \texttt{true and}\ B_2, R\rangle \to \langle B_2, R\rangle$$

$$\cdot\langle \texttt{false and}\ B_2, R\rangle \to \langle \texttt{false}, R\rangle$$

$$\cdot\langle \texttt{not}\ B, R\rangle \to \langle \texttt{not}\ B', R'\rangle \ \textbf{if}\ \cdot\langle B, R\rangle \to \langle B', R'\rangle$$

$$\cdot\langle \texttt{not true}, R\rangle \to \langle \texttt{false}, R\rangle$$

$$\cdot\langle \texttt{not false}, R\rangle \to \langle \texttt{true}, R\rangle$$

$$\cdot\langle X \texttt{:=} A, R\rangle \to \langle X \texttt{:=} A', R'\rangle \ \textbf{if}\ \cdot\langle A, R\rangle \to \langle A', R'\rangle$$

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

$$\cdot\langle St_1; St_2, R\rangle \to \langle St_1'; St_2, R'\rangle \ \textbf{if}\ \cdot\langle St_1, R\rangle \to \langle St_1', R'\rangle$$

$$\cdot\langle \texttt{skip}; St_2, R\rangle \to \langle St_2, R\rangle$$

$$\cdot\langle \{St\}, R\rangle \to \langle St, R\rangle$$

$$\cdot\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2, R\rangle \to \langle \texttt{if}\ B'\ \texttt{then}\ St_1\ \texttt{else}\ St_2, R'\rangle \ \textbf{if}\ \cdot\langle B, R\rangle \to \langle B', R'\rangle$$

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

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

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

$$\cdot\langle St.A, R\rangle \to \langle St'.A, R'\rangle \ \textbf{if}\ \cdot\langle St, R\rangle \to \langle St', R'\rangle$$

$$\cdot\langle \texttt{skip}.A, R\rangle \to \langle \texttt{skip}.A', R'\rangle \ \textbf{if}\ \cdot\langle A, R\rangle \to \langle A', R'\rangle$$

Table 7
R_{MSOS} rewriting logic theory

In a context reduction semantics of a language, one typically starts by defining the syntax of *evaluation contexts*. An evaluation context is a program with a "hole", the hole being a placeholder where the next computational step takes place. If C is such a context and E is some expression whose type fits into the

$$
\begin{array}{rcl}
\mathit{CConf} & ::= & \langle \mathit{CPgm}, \mathit{Store} \rangle \\
\mathit{CPgm} & ::= & [] \mid \texttt{skip}.\mathit{CAExp} \mid \mathit{CStmt}.\mathit{AExp} \\
\mathit{CStmt} & ::= & [] \mid \mathit{CStmt}; \mathit{Stmt} \mid X \texttt{:=} \mathit{CAExp} \mid \texttt{if}\ \mathit{CBExp}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \\
 & \mid & \texttt{halt}\ \mathit{CAExp} \\
\mathit{CBExp} & ::= & [] \mid \mathit{Int} \texttt{<=} \mathit{CAExp} \mid \mathit{CAExp} \texttt{<=} \mathit{AExp} \mid \mathit{CBExp}\ \texttt{and}\ \mathit{BExp} \mid \texttt{not}\ \mathit{CBExp} \\
\mathit{CAExp} & ::= & [] \mid \mathit{Int} + \mathit{CAExp} \mid \mathit{CAExp} + \mathit{AExp}
\end{array}
$$

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

$I_1 + I_2 \to (I_1 +_{Int} I_2)$

$\langle P, \sigma \rangle[X] \to \langle P, \sigma \rangle[(\sigma(X))]$

$\langle P, \sigma \rangle[\texttt{++}X] \to \langle P, \sigma[I/X] \rangle[I]$ 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 8
The *CxtRed* language definition

type of the hole of C, then $C[E]$ is the program formed by replacing the hole of C by E. The characteristic reduction step underlying context reduction is:

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

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

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

By making the evaluation context explicit and changeable, context reduction is, in our view, a significant improvement over small-step SOS. In particular, one can now define control-intensive statements like `halt` *modularly* and at the desired level of computational granularity. Even though the definition in Table 8 gives one the feeling that evaluation contexts and their instantiation come "for free", the application of the "rewrite in context" rule presented above can be expensive in practice. This is because one needs either to parse/search the entire configuration to put it in the form $C[E]$ for some appropriate C satisfying the grammar of evaluation contexts, or to maintain enough information in some special data-structures to perform the split $C[E]$ using only local information and updates. Moreover, this "matching-modulo-the-CFG-of-evaluation-contexts" step needs to be done at every computation step during the execution of a program, so it may easily become the major bottleneck of an executable engine based on context reduction. *Direct* implementations of context reduction such as `PLT-Redex` cannot avoid paying a significant performance penalty [97]. Danvy and Nielsen propose in [27] a technique for efficiently interpreting a restricted form of reduction semantics definitions by means of "refocusing" functions which yield efficient abstract machines. Although these refocusing functions are equationally definable, since we aim here to achieve minimal representational distance, we prefer to translate the definitions into rewriting logic by leaving the rules unchanged and implementing the decompose and plug functions from reduction semantics by means of equations. Next section will present an abstract-machine definition of a programming language in rewriting logic, resembling Felleisen's CK machine [34], which is obtained by applying Danvy and Nielsen's technique.

Context reduction is trickier to faithfully capture as a rewrite theory, since rewriting logic, by its locality, always applies a rule *in* context, without actually having the capability of changing the given context. Also, from a rewriting

$$s2c(\langle P, S\rangle) = \langle C, S\rangle[R] \,\mathbf{if}\, C[R] = s2c(P)$$

$$s2c(\texttt{skip}.I) = [][\texttt{skip}.I]$$
$$s2c(\texttt{skip}.A) = (\texttt{skip}.C)[R] \,\mathbf{if}\, C[R] = s2c(A)$$
$$s2c(St.A) = (C.A)[R] \,\mathbf{if}\, C[R] = s2c(St)$$

$$s2c(\texttt{halt}\ I) = [][\texttt{halt}\ I]$$
$$s2c(\texttt{halt}\ A) = (\texttt{halt}\ C)[R] \,\mathbf{if}\, C[R] = s2c(A)$$
$$s2c(\texttt{while}\ B\ St) = [][\texttt{while}\ B\ St]$$
$$s2c(\texttt{if}\ T\ \texttt{then}\ St_1\ \texttt{else}\ St_2) = [][\texttt{if}\ T\ \texttt{then}\ St_1\ \texttt{else}\ St_2]$$
$$s2c(\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2) = (\texttt{if}\ C\ \texttt{then}\ St_1\ \texttt{else}\ St_2)[R] \,\mathbf{if}\, C[R] = s2c(B)$$
$$s2c(\{St\}) = [][\{St\}]$$
$$s2c(\texttt{skip}; St_2) = [][\texttt{skip}; St_2]$$
$$s2c(St_1; St_2) = (C; St_2)[R] \,\mathbf{if}\, C[R] = s2c(St_1)$$
$$s2c(X\texttt{:=}I) = [][X\texttt{:=}I]$$
$$s2c(X\texttt{:=}A) = (X\texttt{:=}C)[R] \,\mathbf{if}\, C[R] = s2c(A)$$

$$s2c(I_1\texttt{<=}I_1) = [][I_1\texttt{<=}I_2]$$
$$s2c(I\texttt{<=}A) = (I\texttt{<=}C)[R] \,\mathbf{if}\, C[R] = s2c(A)$$
$$s2c(A_1\texttt{<=}A_2) = (C\texttt{<=}A_2)[R] \,\mathbf{if}\, C[R] = s2c(A_1)$$
$$s2c(T\ \texttt{and}\ B_2) = [][T\ \texttt{and}\ B_2]$$
$$s2c(B_1\ \texttt{and}\ B_2) = (C\ \texttt{and}\ B_2)[R] \,\mathbf{if}\, C[R] = s2c(B_1)$$
$$s2c(\texttt{not}\ T) = [][\texttt{not}\ T]$$
$$s2c(\texttt{not}\ B) = (\texttt{not}\ C)[R] \,\mathbf{if}\, C[R] = s2c(B)$$

$$s2c(X) = [][X]$$
$$s2c(\texttt{++}X) = [][\texttt{++}X]$$
$$s2c(I_1 + I_2) = [][I_1 + I_2]$$
$$s2c(I + A) = (I + C)[R] \,\mathbf{if}\, C[R] = s2c(A)$$
$$s2c(A_1 + A_2) = (C + A_2)[R] \,\mathbf{if}\, C[R] = s2c(A_1)$$

Table 9
Equational definition of *s2c*

point of view, context-reduction captures context-sensitive rewriting, which, although supported by rewriting logic in the form of congruence restricted to the non-frozen arguments of each operator, cannot be captured "as-is" in its full generality within rewriting logic.

$c2s([][H]) = H$
$c2s(\langle P, S\rangle[H]) = \langle c2s(P[H]), S\rangle$
$c2s(\langle I\rangle[H]) = \langle I\rangle$
$c2s(E_1.E_2[H]) = c2s(E_1[H]).c2s(E_2[H])$
$c2s(\texttt{halt}\ E[H]) = \texttt{halt}\ c2s(E[H])$
$c2s(\texttt{while}\ E_1\ E_2[H]) = \texttt{while}\ c2s(E_1[H])\ c2s(E_2[H])$
$c2s(\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2[H]) = \texttt{if}\ c2s(E[H])\ \texttt{then}\ c2s(E_1[H])\ \texttt{else}\ c2s(E_2[H])$
$c2s(\{E\}[H]) = \{c2s(E[H])\}$
$c2s(E_1; E_2[H]) = c2s(E_1[H]); c2s(E_2[H])$
$c2s(X\texttt{:=}E[H]) = X\texttt{:=}c2s(E[H])$
$c2s(\texttt{skip}[H]) = \texttt{skip}$
$c2s(E_1\texttt{<=}E_2[H]) = c2s(E_1[H])\texttt{<=}c2s(E_2[H])$
$c2s(E_1\ \texttt{and}\ E_2[H]) = c2s(E_1[H])\ \texttt{and}\ c2s(E_2[H])$
$c2s(\texttt{not}\ E[H]) = \texttt{not}\ c2s(E[H])$
$c2s(\texttt{true}[H]) = \texttt{true}$
$c2s(\texttt{false}[H]) = \texttt{false}$
$c2s(\texttt{++}X[H]) = \texttt{++}X$
$c2s(E_1 + E_2[H]) = c2s(E_1[H]) + c2s(E_2[H])$
$c2s(I[H]) = I$

Table 10
Equational definition of *c2s*

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

$$\cdot(I_1 + I_2) \to (I_1 +_{Int} I_2)$$
$$\cdot(\langle P, S\rangle[X]) \to \langle P, S\rangle[(S[X])]$$
$$\cdot(\langle P, S\rangle[\texttt{++}X]) \to \langle P, S[X \leftarrow I]\rangle[I] \ \textbf{if}\ I = s(S[X])$$

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

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

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

$$\cdot(C[R]) \to C[R'] \ \textbf{if}\ \cdot(R) \to R'$$
$$\cdot(Cfg) \to c2s(C[R]) \ \textbf{if}\ \cdot(s2c(Cfg)) \to C[R]$$

$$eval(P) = reduction(\langle P, \emptyset\rangle)$$
$$reduction(Cfg) = reduction(\cdot(Cfg))$$
$$reduction(\langle I\rangle) = I$$

Table 11
$\mathcal{R}_{CxtRed}$ rewriting logic theory

interpreting context-reduction definitions, and it is a morphism on the syntax. Notice that (from the defining equations) we have the guarantee that it will be applied only to "well-formed" contexts (i.e., contexts containing only one hole). The rewrite theory $\mathcal{R}_{CxtRed}$ is obtained by adding the rules in Table 11 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 $\texttt{++}x$**:** $\texttt{++}x$ parses as $[][\texttt{++}x]$. Also $\mathcal{R}_{CxtRed} \vdash s2c(\texttt{++}x) = [][\texttt{++}x]$ in one step (it is an instance of an axiom).
Case $a_1\texttt{<=}a_2$**:** 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
$\mathcal{R}_{CxtRed} \vdash s2c(a_1\texttt{<=}a_2) = (a_1\texttt{<=}c)[r]$.
Case $x\texttt{:=}a$**:** $x\texttt{:=}a$ parses as $[][x\texttt{:=}a]$ iff $a \in Int$, iff
$\mathcal{R}_{CxtRed} \vdash s2c(x\texttt{:=}i) = [][x\texttt{:=}i]$.
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
$\mathcal{R}_{CxtRed} \vdash s2c(at.a) = st.c[r]$.
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
$\mathcal{R}_{CxtRed} \vdash s2c(p) = c'[r]$ and $c = \langle c', s\rangle$ iff
$\mathcal{R}_{CxtRed} \vdash s2c(\langle p, s\rangle) = \langle c', s\rangle[r]$.
(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 $CxtRed \vdash \langle p, \sigma\rangle \to \langle p', \sigma\rangle$. We only discuss some cases; the others are treated similarly.

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

$CxtRed \vdash \langle p, \sigma\rangle \to \langle p', \sigma\rangle$ 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
$\mathcal{R}_{CxtRed} \vdash s2c(\langle p, s\rangle) = c[\texttt{not true}]$ and $\mathcal{R}_{CxtRed} \vdash c2s(c[\texttt{false}]) = \langle p', s\rangle$ 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 $\mathcal{R}_{CxtRed} \vdash \cdot(\texttt{not true}) \to^1 \texttt{false}$).

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

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

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

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

Weaknesses. It still only allows "interleaving semantics" for concurrency. Although context-sensitive rewriting might seem to be easily implementable by rewriting, in fact all current implementations of context reduction work by transforming context grammar definitions into traversal functions, thus being as (in)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). However, one might obtain efficient implementations for restricted forms of context-reduction definitions by applying refocusing techniques [27].

9 A Continuation-Based Semantics

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

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

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

$aexp(I) = I$

$aexp(A_1 + A_2) = (aexp(A_1), aexp(A_2)) \curvearrowright +$

$k(aexp(X) \curvearrowright K)\ store(Store) \to k(Store[X] \curvearrowright K)\ store(Store)$

$k(aexp(\texttt{++}X) \curvearrowright K)\ store((X = I)\, Store)$
$\to k(s(I) \curvearrowright K)\ store((X = s(I))\, Store)$

$k(I_1, I_2 \curvearrowright + \curvearrowright K) \to k(I_1 +_{Int} I_2 \curvearrowright K)$

$bexp(true) = true$ $\qquad$ $bexp(false) = false$

$bexp(A_1 \texttt{<=} A_2) = (aexp(A_1), aexp(A_2)) \curvearrowright \leq$

$bexp(B_1\ \texttt{and}\ B_2) = bexp(B_1) \curvearrowright and(bexp(B_2))$

$bexp(\texttt{not}\ B) = bexp(B) \curvearrowright not$

$k(I_1, I_2 \curvearrowright \leq \curvearrowright K) \to k(I_1 \leq_{Int} I_2 \curvearrowright K)$

$k(true \curvearrowright and(K_2) \curvearrowright K) \to k(K_2 \curvearrowright K)$

$k(false \curvearrowright and(K_2) \curvearrowright K) \to k(false \curvearrowright K)$

$k(T \curvearrowright not \curvearrowright K) \to k(not_{Bool} T \curvearrowright K)$

$stmt(\texttt{skip}) = nothing$

$stmt(X := A) = aexp(A) \curvearrowright write(X)$

$stmt(St_1; St_2) = stmt(St_1) \curvearrowright stmt(St_2)$ $\qquad$ $stmt(\{St\}) = stmt(St)$

$stmt(\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2) = bexp(B) \curvearrowright if(stmt(St_1), stmt(St_2))$

$stmt(\texttt{while}\ B\ St) = bexp(B) \curvearrowright while(bexp(B), stmt(St))$

$stmt(\texttt{halt}\ A) = aexp(A) \curvearrowright halt$

$k(I \curvearrowright write(X) \curvearrowright K)\ store(Store) \to k(K)\ store(Store[X \leftarrow I])$

$k(true \curvearrowright if(K_1, K_2) \curvearrowright K) \to k(K_1 \curvearrowright K)$

$k(false \curvearrowright if(K_1, K_2) \curvearrowright K) \to k(K_2 \curvearrowright K)$

$k(true \curvearrowright while(K_1, K_2) \curvearrowright K) \to k(K_2 \curvearrowright K_1 \curvearrowright while(K_1, K_2) \curvearrowright K)$

$k(false \curvearrowright while(K_1, K_2) \curvearrowright K) \to k(K)$

$k(I \curvearrowright halt \curvearrowright K) \to k(I)$

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

$\langle P \rangle = result(k(pgm(P))\ store(empty))$

$result(k(I)\ store(Store)) = I$

Table 12
Rewriting logic theory $\mathcal{R}_K$ (continuation-based definition of the language)

rules exactly encode the computation steps semantically, yielding the intended computational granularity. Specifically *pgm*, *stmt*, *bexp*, *aexp* are used to flatten the program to a continuation, taking into account the order of evaluation. The continuation is defined as a list of tasks, where the list constructor "$\_ \curvearrowright \_$" 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 12. Lists of expressions are evaluated using the following (equationally defined) mechanism:

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

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

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

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

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

Strengths. In continuation-based semantics there is no need to search for a redex anymore, because the redex is always at the top. It is much more efficient than *direct* implementations of evaluation contexts or small-step SOS.

Also, this style greatly reduces the need for conditional rules/equations; conditional rules/equations might involve inherently inefficient reachability analysis to check the conditions and are harder to deal with in parallel environments. An important "strength" specific to the rewriting logic approach is that reductions can now apply wherever they match, in a *context-insensitive* way. Additionally, continuation-based definitions in the RLS style above are very modular (particularly due to the use of matching modulo associativity and commutativity).

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

Relation with Context Reduction

We next show the equivalence between the continuation-based and the context reduction rewriting logic definitions. The specification in Table 13 relates the two semantics, showing that at each computational "point" it is possible to extract from our continuation structure the current expression being evaluated. For each syntactical construct $Syn \in \{AExp, BExp, Stmt, Pgm\}$, 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 remainder 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 13.

We will show below that for any step *CxtRed* takes, $\mathcal{R}_K$ 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 13
Recovering the abstract syntax trees from continuations

is no need for that when using continuations). Also, no steps will be performed for loop unrolling, because this is *not* a computational step; it is a straightforward structural equivalence. In fact, note that, because of its incapacity to distinguish between computational steps and structural equivalences, *CxtRed*

does not capture the intended granularity of `while`: it wastes a computation step for unrolling the loop and one when dissolving the while into skip; neither of these steps has any computational content.

In order to clearly explain the relation between reduction contexts and continuations, we go a step further and define a new rewrite theory $\mathcal{R}_{K'}$ 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*:

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

replacing the equation for evaluating lists of expressions, namely,

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

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

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

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 $\mathcal{R}_K$, $Tree_{\mathcal{R}_K}(t)$ is bisimilar to $Tree_{\mathcal{R}_{K'}}(t)$.

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

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

Proof.

$\texttt{++}x = [][\texttt{++}x]$: $\mathcal{R}_{K''} \vdash k(aexp(\texttt{++}x)) = k(aexp(\texttt{++}x) \curvearrowright aexp([]))$
$a_1 + a_2 = [] + a_2[a_1]$: $\mathcal{R}_{K''} \vdash k(aexp(a_1 + a_2)) = k((aexp(a_1), aexp(a_2)) \curvearrowright +)$
$= k(aexp(a_1) \curvearrowright (aexp([]), aexp(a_2)) \curvearrowright +) = k(aexp(a_1) \curvearrowright aexp([] + a_2))$
$i_1 + a_2 = i_1 + [][a_2]$: $\mathcal{R}_{K''} \vdash k(aexp(i_1 + a_2)) = k((aexp(i_1), aexp(a_2)) \curvearrowright +)$
$= k(aexp(a_2) \curvearrowright (i_1, aexp([])) \curvearrowright +) = k(aexp(a_2) \curvearrowright aexp(i_1 + [])).$
b_1 `and` $b_2 = []$ `and` $b_2[b_1]$:
$\mathcal{R}_{K''} \vdash k(bexp(b_1$ `and` $b_2)) = k(bexp(b_1) \curvearrowright and(bexp(b_2)))$
$= k(bexp(b_1) \curvearrowright bexp([]) \curvearrowright and(aexp(b_2))) = k(bexp(b_1) \curvearrowright bexp([]$`and` $b_2)).$
t `and` $b_2 = [][t$ `and` $b_2]$:
$\mathcal{R}_{K''} \vdash k(bexp(t$ `and` $b_2)) = k(bexp(t$ `and` $b_2) \curvearrowright bexp([])).$

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

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 CxtRed $\vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ then $\mathcal{R}_{K'} \vdash k(pgm(p))\ store(s) \to^{\leq 1} k(pgm(p'))\ store(s')$ and $s' \simeq \sigma'$, where $\to^{\leq 1} = \to^0 \cup \to^1$.
(2) If $\mathcal{R}_{K'} \vdash k(pgm(p))\ store(s) \to k(k')\ store(s')$ then there exists p' and σ' such that CxtRed $\vdash \langle p, \sigma\rangle \to^* \langle p', \sigma'\rangle$, $\mathcal{R}_{K'} \vdash k(pgm(p')) = k(k')$ and $s' \simeq \sigma'$.
(3) CxtRed $\vdash \langle p, \bot\rangle \to^* i$ iff $\mathcal{R}_{K'} \vdash \langle p\rangle \to i$ for any $p \in Pgm$ and $i \in Int$.

Proof. (Sketch)

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

□

10 The Chemical Abstract Machine

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

The basic metaphor giving its name to the Cham is inspired by Banâtre and Le Métayer's GAMMA language [6]. It views a distributed state as a "solution" in which many "molecules" float, and understands concurrent transitions as "reactions" that can occur simultaneously in many points of the solution. It is possible to define a variety of chemical abstract machines. Each of them corresponds to a rewrite theory satisfying certain common conditions.

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

sorts Molecule, Molecules, Solution .
subsorts Solution < Molecule < Molecules .
op $\lambda : \longrightarrow$ *Molecules .*
op $\_,\_$: *Molecules Molecules* $\longrightarrow$ *Molecules .*
op $\{|\_|\}$: *Molecules* $\longrightarrow$ *Solution* . `*** membrane operator`
op $\_ \triangleleft \_$: *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 [8]. *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

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

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

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

The *CHAM* definition for our simple programming language takes the *CxtRed* definition in Table 8 as a starting point. More precisely, we follow the "refocusing" technique [27]. We distinguish two kinds of molecules: syntactic molecules and store molecules. Syntactic molecules are either language constructs or evaluation contexts and we will use "$[\_ \mid \_]$" as a molecule constructor for stacking molecules. We let C range over syntactic molecules representing stacked con-

texts. Store molecules are pairs (x, i), where x is a variable and i is an integer. The store is a solution containing store molecules. Then the definition of "refocusing" functions is translated into heating/cooling rules, bringing the redex to the top of the syntactic molecule. This allows for the reduction rules to only operate at the top, in a conceptually identical way as for continuation based definitions in Table 12, both of them following the same methodology introduced in [72].

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

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

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

$$\texttt{skip}; 5/(2/x) \rightharpoonup \{\!|5/(2/x) \rhd \{\!|\texttt{skip}; []|\!\}\!\} \rightharpoonup \{\!|2/x \rhd \{\!|5/[] \rhd \{\!|\texttt{skip}; []|\!\}\!\}\!\}$$
$$\rightharpoonup \{\!|x \rhd \{\!|2/[] \rhd \{\!|5/[] \rhd \{\!|\texttt{skip}; []|\!\}\!\}\!\}\!\}$$

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

$$\{\!|x \rhd \{\!|2/[] \rhd \{\!|5/[] \rhd \{\!|\texttt{skip}; []|\!\}\!\}\!\}\!\} \rightharpoondown \{\!|x \rhd \{\!|2/[] \rhd \{\!|5/[], \texttt{skip}; []|\!\}\!\}\!\}$$
$$\rightharpoondown \{\!|x \rhd \{\!|2/[], 5/[], \texttt{skip}; []|\!\}\!\} \rightharpoonup \{\!|x \rhd \{\!|5/[] \rhd \{\!|2/[], \texttt{skip}; []|\!\}\!\}\!\}$$
$$\rightharpoonup \{\!|x \rhd \{\!|5/[] \rhd \{\!|2/[] \rhd \{\!|\texttt{skip}; []|\!\}\!\}\!\}\!\}$$

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

$$\{\!|x \rhd \{\!|5/[] \rhd \{\!|2/[] \rhd \{\!|\texttt{skip}; []|\!\}\!\}\!\}\!\} \rightharpoondown \{\!|5/x \rhd \{\!|2/[] \rhd \{\!|\texttt{skip}; []|\!\}\!\}\!\}$$
$$\rightharpoondown \{\!|2/(5/x) \rhd \{\!|\texttt{skip}; []|\!\}\!\} \rightharpoondown \texttt{skip}; 2/(5/x)$$

However, $\texttt{skip}; 5/(2/x)$ and $\texttt{skip}; 2/(5/x)$ are obviously *not* structurally equivalent.

The above language definition does not exhibit the strengths of the Cham,

$St.A \rightleftharpoons [St \mid [].A]]$

$\texttt{skip}.A \rightleftharpoons [A \mid [\texttt{skip}.[]]]$

$[X\texttt{:=}A \mid C] \rightleftharpoons [A \mid [X\texttt{:=}[] \mid C]]$

$[St_1; St_2 \mid C] \rightleftharpoons [St_1 \mid [[]; St_2 \mid C]]$

$[\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2 \mid C] \rightleftharpoons [B \mid [\texttt{if}\ []\ \texttt{then}\ St_1\ \texttt{else}\ St_2 \mid C]]$

$[\texttt{halt}\ A \mid C] \rightleftharpoons [A \mid [\texttt{halt}\ [] \mid C]]$

$[A_1\texttt{<=}A_2 \mid C] \rightleftharpoons [A_1 \mid []\texttt{<=}A_2 \mid C]]$

$[I\texttt{<=}A \mid C] \rightleftharpoons [A \mid [I\texttt{<=}[] \mid C]]$

$[B_1\ \texttt{and}\ B_2 \mid C] \rightleftharpoons [B_1 \mid [[]\ \texttt{and}\ B_2 \mid C]]$

$[\texttt{not}\ B \mid C] \rightleftharpoons [B \mid [\texttt{not}\ [] \mid C]]$

$[A_1 + A_2 \mid C] \rightleftharpoons [A_1 \mid [] + A_2 \mid C]]$

$[I + A \mid C] \rightleftharpoons [A \mid [I + [] \mid C]]$

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

$[X \mid C], \{\!|(X, I) \rhd \sigma|\!\} \rightarrow [I \mid C], \{\!|(X, I) \rhd \sigma|\!\}$

$[\texttt{++}X \mid C], \{\!|(X, I) \rhd \sigma|\!\} \rightarrow [I +_{Int} 1 \mid C], \{\!|(X, I +_{Int} 1) \rhd \sigma|\!\}$

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

$\texttt{true}\ \texttt{and}\ B \rightarrow B$

$\texttt{false}\ \texttt{and}\ B \rightarrow \texttt{false}$

$\texttt{not}\ \texttt{true} \rightarrow \texttt{false}$

$\texttt{not}\ \texttt{false} \rightarrow \texttt{true}$

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

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

$\texttt{skip}; St \rightarrow St$

$\{St\} \rightarrow St$

$[X\texttt{:=}I \mid C], \{\!|(X, I') \rhd \sigma|\!\} \rightarrow [\texttt{skip} \mid C], \{\!|(X, I) \rhd \sigma|\!\}$

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

$[\texttt{halt}\ I \mid C], \sigma \rightarrow I$

$\texttt{skip}.I, \sigma \rightarrow I$

Table 14
The *CHAM* language definition

since Cham was designed to handle easily concurrent constructs, which are missing from our language. However, making the above language concurrent in Cham comes at no additional effort. One can execute multiple programs at the same time, sharing the store, simply by putting them together, and together with the store at the top-level solution and replacing the rule for the end of the program by $\texttt{skip}.I \to I$, to allow all programs to finish their evaluation and keep the results.

When Cham definitions follow the style in Table 14, i.e., taking a context-reduction-like approach, one could use as evaluation strategies *heating only on redexes* and *cooling only on values*, which would lead to a deterministic abstract-machine. Moreover, one can notice that airlock rules were introduced to select elements from a set without specifying the rest of the set, abstracted by a molecule. Efficient implementations should probably do exactly the opposite, that is, matching in the sets. To do that in our rewrite framework, one would orient the airlock rules in the sense of inserting back the "airlocked" molecules into their original solution and to apply them on the terms of the existing rules, to make the definition executable. The only rules changing in the definition above are those involving the store; for example, the assignment rule is transformed into:

$$[X\,\texttt{:=}I \mid C], \{\!|(X, I'), \sigma|\!\} \to [\texttt{skip} \mid C], \{\!|(X, I), \sigma|\!\}$$

One should notice that the specification obtained by these transformations is equivalent to the initial one, since it does not change the equivalence classes and the transitions. The main advantage of the newly obtained specification is that it is also executable in a deterministic fashion, that is, there is no need to search for a final state anymore.

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

Weaknesses. Heating/cooling rules are hard to implement efficiently in general —an implementation allowing them to be bidirectional in an uncontrolled manner would have to *search* for final states, possibly leading to a combinatorial explosion. Rewriting strategies such as those in [10, 29, 93] can be of help for solving particular instances of this problem. Although this solution-molecule paradigm seems to work pretty well for languages in which the structure of the state is simple enough, it is not clear how one could represent the state for complex languages, with threads, locks, environments, an so on. Finally, Chams provide no mechanism to freeze the current molecular structure as a "value", and then to store or retrieve it, as we would need in order to define

language features like call/cc. Even though it was easy to define halt because we simply discarded the entire solution, it would seem hard or impossible to define more complex control-intensive language features in Cham.

11 Experiments

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

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

We compare the running times and memory requirements of the interpreters derived mechanically using the above-mentioned procedures, with those of the "free" interpreters given by executing the definition "as-is" on two rewrite engines (marked with $\star$ in the tables), namely ASF+SDF `1.5` (a compiler) and `Maude 2.2` (a fast interpreter with good tool support), as well as with those obtained executing off-the-shelf interpreter implementations in Scheme, used in teaching programming languages (marked with $\sharp$ in the tables). For Scheme we have used PLT-Scheme as an interpreter and language interpreter implementations from [35], chapters 3.9 (evaluation semantics) and 7.3 (continuation based semantics), and a PLT-Redex definition given as example in the installation package (for context reduction). Big-step interpreters are also compared against bc, a C-written interpreter for a subset of C working only with integers (bc comes as part of UNIX; type "man bc" for documentation), and two interpreters implemented using monads in Haskell and Ocaml

(we mark these interpreters with ♭ in Table 16). Since RLS representations of MSOS and Cham definitions rely intensively on matching modulo associativity and commutativity, which is only supported by Maude, we have only performed some experiments on their RLS definitions in Maude. For Cham we preferred to give the times obtained by using the novel transformations and strategies presented in Section 10 for making the specification "more executable". Using the specification *as is*, Cham is extremely ineffective when executed: it takes about 1205MB of memory and 188 seconds to search for the solution of running the Collatz program (explained below) up to 3.

One may naturally ask: "What is the point of all these experiments? They show little or nothing to support the RLS resulting definitions compared to their original definitions, and only show what programs (interpreters) in what programming languages are more efficient than others." Our goal here is to convey the reader our strong belief, supported by empirical evaluation, that the working language designer may be better off in practice *formally defining* a desired language, using some preferred definitional style, than *implementing an interpreter in an ad-hoc way* for that language, even in a preferred programming language. Unfortunately, the latter approach is also how programming language concepts are being taught in many places. Formal definitions tend to be significantly more compact, easier to read and more modular than ad-hoc language implementations, so they are easier to change and experiment with. Additionally, they can serve as a mathematical object capturing the essence of the desired language. One can then use this mathematical object for many other purposes in addition to executing programs, including formal analyses such as theorem proving and model-checking, static analysis, partial evaluation, compiler generation, and so on. Of course, this belief transcends the boundaries of rewriting logic; what RLS gives us here is a unified framework, with a uniform notation supported by a rigorous computational logic, in which one can formally define programming languages using any of the desired styles. None of the translations from RLS definitions into programming languages has been implemented, because that is not the focus of this paper. Nevertheless, we strongly believe that they can be implemented with relatively little effort.

One of the programs chosen to test various implementations consists of n nested loops, each of 2 iterations, parameterized by n. The other program tests the Collatz's conjecture up to 300. Collatz's conjecture states that starting from any positive number n and performing the following operations:

- if n is even then divide it by 2;
- if n is odd then multiply it by 3 and add 1;

after a finite number of steps, the value of n will become 1. To make the program more computation-intensive (and also to maximize the number of language constructs used), we here use repeated subtraction to compute divi-

```
x0 := 0;
while (++x0<=2){
  x1:=0;
  while (++x1<=2){
      ...
          x18:=0;
          while (++x18<=2){
            skip;
          }
      ...
  }
}.0
```

(a)

```
nr:=300;
while (not (nr<=2)){
  n:=nr;
  nr:=nr − 1;
  while (not (n==1)){
    steps:=steps + 1;
    r:=n;
    q:=0;
    while (not (r<=1)){
      r:=r − 2;
      q:=q + 1
    };
    if (r==0)
       then n:=q
       else n:=3 ∗ n + 1
    }
  }
}.steps
```

(b)

Table 15
Programs used in evaluation: (a) A tower of loops, each performing two iterations; (b) Program testing Collatz's conjecture up to 300.

sion. We also count in `steps` the cumulative number of operations performed until 1 is reached for all numbers tested and return it as the result of the program. The source code for the programs used is presented in Table 15.

Tables 16, 17, 18, and 19, give for each definitional style the running time of the various interpreters. For the largest number n (18) of nested loops, peak memory usage was also recorded. Times are expressed in seconds. A limit of 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 6.4.2` compiler. For Ocaml we have used the `ocamlcopt 3.09.3` compiler. For Prolog we have compiled the programs using the `gprolog 1.3.0` compiler. For Scheme we have used the PLT-Scheme (`mzscheme 3.7.1`) inter-

	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 [35]	3.8	7.4	30.2	13mb	122.3

Table 16
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 17
Execution times for Small Step definitions

preter. Tests were performed on an Intel Pentium 4@2GHz with 1GB RAM, running Linux.

To have an overview of execution times obtained by using the RLS definition *as is* for all the styles presented, Table 20 shows, side by side, their execution times in Maude.

Prolog yields pretty fast interpreters. However, for backtracking reasons, it needs to maintain the stack of all predicates tried on the current path, thus the amount of memory grows with the number of computational steps. The style promoted in [35] seems to also take into account efficiency. Its only drawback

	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	0.8	76.2	162.8	713.2	6mb	1931.6
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 18
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 [35]	5.9	11.3	45.2	10mb	323.6

Table 19
Execution times for Continuation based definitions

	N nested loops(1..2)				Collatz' conjecture
N	15	16	18	Memory for 18	up to 300
Big-Step	3.8	7.7	31.5	6mb	184.5
Small-Step	63.4	131.2	597.4	6mb	1249.1
Context-Reduction	76.2	162.8	713.2	6mb	1931.6
Continuation-Based	8.4	15.6	63.2	7mb	483.9
MSOS	61.9	127.4	566.3	6mb	1421.5
Cham	15.7	31.5	129.2	6mb	618.0

Table 20
Execution times for RLS definitions interpreted in Maude

is the fact that it looks more like an implementation, the representational distance to the big-step definition being much bigger than in interpreters based on RLS. The PLT-Redex implementation of context reduction seems to serve more a didactic purpose. It compensates for lack of speed by providing a nice interface and the possibility to visually trace a run. The rewriting logic implementations seem to be quite efficient in terms of speed and memory usage, while keeping a minimal representational distance to the operational semantics definitions. In particular, RLS definitions interpreted in Maude are comparable in terms of efficiency with the interpreters in Scheme, while having the advantage of being formal definitions. The main reason for Maude and Scheme being slower than the others, is because they are both interpreters while the others are compilers. It is well known that compilers usually generate executables one order of magnitude faster than their interpreted versions. Also, it is good to notice that the interpreter obtained by mechanically compiling the RLS definition in Ocaml can reach the speed of the hand-optimized, C-written `bc` interpreter.

12 Related Work

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

Algebraic denotational semantics. This approach, (see [14, 39, 62, 95] for early papers and [37, 88] for two more recent books), is a special case of RLS, namely, the case in which the rewrite theory $\mathcal{R}_{\mathcal{L}}$ 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, 39]), 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 [3–5, 12, 13, 17, 18, 21, 24, 26, 30–33, 36, 42, 43, 45, 53, 55, 56, 73, 75, 76, 84–86, 89–91]. A first snapshot of the RLS project was given in [56], and a second in [57]. This paper can be viewed as third snapshot focusing on the variety of definitional styles supported. In

particular, a substantial body of experience in giving programming language definitions, and using those definitions both for execution and for analysis purposes has already been gathered. For example, Java 1.4 (see also [20] for a complete formal semantics) and the JVM (see [30, 33]) 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, 33]. A semantics of a Caml-like language with threads was discussed in detail in [56], and a modular rewriting logic semantics of a subset of CML has been given in [18] using the Maude MSOS tool [19]. A definition of the Scheme language has been given in [26]. Other language case studies, all specified in Maude, include BC [13], CCS [13, 90], CIAO [85], Creol [43], ELOTOS [89], MSR [16, 83], PLAN [84, 85], the ABEL hardware description language [45], SILF [42], FUN [72], Orc [4, 5], and the π-calculus [86].

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

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

Abstract state machines. Abstract State Machine (ASM) [40] can encode any computation and have a rigorous semantics, so any programming language can be defined as an ASM and thus implicitly be given a semantics. Both big- and small-step ASM semantics have been investigated. The semantics of vari-

ous programming languages, including, for example, Java [81], has been given using ASMs. There are interesting connections between ASMs and rewriting logic, but their discussion is beyond the scope of this paper.

13 Conclusions

In this paper we have tried to show how RLS can be used as a logical framework for operational semantics definitions of programming languages. In particular, by showing in detail how it can faithfully capture big-step and small-step SOS, 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 *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, New York, NY, USA, 1990. ACM Press.

[2] G. A. 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)*, volume 153(2) of *Electronic Notes in Theoretical Computer Science*, pages 213–239, 2006.

[3] W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for java verification against a rewriting semantics. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 412–426. Springer, 2005.

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

[5] M. AlTurki and J. Meseguer. Real-time rewriting semantics of orc. In M. Leuschel and A. Podelski, editors, *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wroclaw, Poland*, pages 131–142. ACM Press, 2007.

[6] J.-P. Banâtre and D. L. Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, 1990.

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

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

[9] 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.

[10] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[11] 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.

[12] 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.

[13] C. Braga and J. Meseguer. Modular rewriting semantics in practice. In *Proceedings of the Fifth International Workshop on Rewriting Logic*

and Its Applications (WRLA 2004), volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 393–416. Elsevier, 2005.

[14] M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(1):54–99, 1987.

[15] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[16] 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, *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 183–207. Elsevier, 2005.

[17] 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`.

[18] F. Chalub and C. Braga. A modular rewriting semantics for CML. *The Journal of Universal Computer Science*, 10(7):789–807, 2004.

[19] F. Chalub and C. Braga. Maude MSOS tool. In G. Denker and C. Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 133–146. Elsevier, 2007.

[20] F. Chen and G. Roşu. Rewriting Logic Semantics of Java 1.4, 2004. `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java`.

[21] F. Chen, G. Rosu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 2003.

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

[23] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[24] M. Clavel and J. Santa-Cruz. ASIP + ITP: A verification tool based on algebraic semantics. In *PROLE 2005: V Jornadas sobre Programacin y Lenguajes*, pages 149–158. Thomson, 2005.

[25] D. Clément, J. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In K. Fuchi and M. Nivat, editors, *Proceedings of the France-Japan AI and CS Symposium*, pages 49–89. ICOT, Japan, 1986. Also, Information Processing Society of Japan, Technical Memorandum PL-86-6 and Rapport de recherche #0416, INRIA.

[26] M. d'Amorim and G. Rosu. An equational specification for the Scheme

language. *The 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.

[27] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[28] 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.

[29] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In T. B. d. l. T. M. Archer and C. Muoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006)*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 3–25. Elsevier, 2007.

[30] A. Farzan. *Static and dynamic formal analysis of concurrent systems and languages: a semantics-based approach*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[31] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[32] A. Farzan and J. Meseguer. Partial order reduction for rewriting semantics of programming languages. In G. Denker and C. Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 61–78. Elsevier, 2007.

[33] A. Farzan, J. Meseguer, and G. Rosu. Formal JVM code analysis in JavaFAN. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2004.

[34] 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.

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

[36] A. Garrido, J. Meseguer, and R. Johnson. Algebraic semantics of the

C preprocessor and correctness of its refactorings. Technical Report UIUCDCS-R-2006-2688, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2006.

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

[38] 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.

[39] J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Díaz and I. Ramos, editors, *Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19-25, 1981, Proceedings*, volume 107 of *Lecture Notes in Computer Science*, pages 292–309. Springer, 1981.

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

[41] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[42] M. Hills, T. F. Şerbănuţă, and G. Roşu. A rewrite framework for language definitions and for generation of efficient interpreters. In G. Denker and C. Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 215–231. Elsevier, 2007.

[43] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 375–392. Elsevier, 2005.

[44] G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[45] 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, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 47–60. Elsevier, 2007.

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

[47] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–

343, New York, NY, USA, 1995. ACM Press.

[48] 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.

[49] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

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

[51] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.

[52] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[53] 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.

[54] J. Meseguer. A rewriting logic sampler. In D. V. Hung and M. Wirsing, editors, *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings*, volume 3722 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2005.

[55] J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2004.

[56] J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.

[57] J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[58] D. Miller. Representing and reasoning with operational semantics. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Sci-*

ence, pages 4–20. Springer, 2006.

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

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

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

[62] P. D. Mosses. Unified algebras and action semantics. In B. Monien and R. Cori, editors, *STACS 89, 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Proceedings*, volume 349 of *Lecture Notes in Computer Science*, pages 17–35. Springer, 1989.

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

[64] P. D. Mosses. Pragmatics of modular SOS. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.

[65] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[66] G. Nadathur and D. Miller. An overview of Lambda-PROLOG. In K. A. B. Robert A. Kowalski, editor, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988, Proceedings*, pages 810–827. MIT Press, 1988.

[67] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. In N. Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 285–314. Elsevier, 2005.

[68] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM Press.

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

[70] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

[71] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[72] 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. K was first introduced in 2003, in the techni-

cal report UIUCDCS-R-2003-2897: lecture notes of CS322 (programming language design).

[73] G. Rosu, R. P. Venkatesan, J. Whittle, and L. Leustean. Certifying optimality of state estimation programs. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 301–314. Springer, 2003.

[74] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[75] 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.

[76] R. Sasse and J. Meseguer. Java+ITP: A verification tool based on hoare logic and algebraic semantics. In G. Denker and C. Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 29–46. Elsevier, 2007.

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

[78] 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.

[79] 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.

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

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

[82] M.-O. Stehr. CINNI - a generic calculus of explicit substitutions and its application to lambda-, sigma- and pi- calculi. In K. Futatsugi, editor, *Proceedings of the Third International Workshop on Rewriting Logic and its Applications (WRLA 2000)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[83] 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`, 2004.

[84] M.-O. Stehr and C. L. Talcott. Plan in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 240–260. Elsevier, 2002.

[85] 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.

[86] 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, *Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[87] M. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

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

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

[90] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[91] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.

[92] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[93] 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, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003.

[94] P. Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

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

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

[97] Y. Xiao, Z. M. Ariola, and M. Mauny. From syntactic theories to interpreters: A specification language and its compilation. *The Computing Research Repository (CoRR)*, cs.PL/0009030, September 2000.

A Obtaining interpreters from RLS definitions

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

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

A.1 Big-Step based definitions

The rewriting rule for assignment in big-step is:

$$\langle X := A, S\rangle \to \langle S'[X \leftarrow I]\rangle \quad \textbf{if} \quad \langle A, S\rangle \to \langle I, S'\rangle$$

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

```
[] <I,S1> := <A,S>
   ==========================
   <X := A,S> = bind(S1,X,I)
```

Haskell We use `Scgf(st,s)` and `Acfg(a,s)`, etc., to encode configurations $\langle st, s\rangle$ and $\langle a, s\rangle$, respectively. We define an evaluation function for each type of configuration, for example `eStmt` is the function evaluating `Scfg` configurations and `eAExp` is evaluating `Acfg` configurations. The matching of the evaluation of premises is performed by using the let construct.

```
eStmt (Scfg (Assign x a) s) =
       let (Acfg (Int i) s1) = eAExp (Acfg a s)
           in (bind s1 x i)
```

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

```
rl < X := A,S > => S1[X <- I] if < A,S > => {I,S1} .
```

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

```
let rec eval = function
 ...
  | Scfg(Assign(x,a),s) ->
      (match eval (Acfg(a,s)) with Acfg(Int(i),s1) ->
                                              (bind s1 x i))
```

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

```
eStmt(scfg(X = A,S),S2) :- eAExp(acfg(AE,S),acfg(I,S1)),
                           bind(S1,X,I,S2).
```

A.2 Continuation-based definitions

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

$$stmt(X := A) = aexp(A) \curvearrowright write(X)$$
$$k(I \curvearrowright write(X) \curvearrowright K)\ store(Store) \rightarrow k(K)\ store(Store[X \leftarrow I])$$

ASF+SDF Again, the translation to ASF+SDF implies minimal or no modifications. Note that ASF+SDF makes no distinction between equations and rules, all of them being written as equations.

```
[] stmt(X := A) = aexp(A) -> write(X)
[] k(int(I) -> write(X) -> K) store(Store)
 = k(K) store(bind(Store,X,I))
```

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

```
stmt (Assign x a)) = (aexp a) ++ [Kwrite x]
result (Kval (Vint i):Kwrite x:k) s = result k (bind s x i)
```

Maude Representation in Maude is the exact rewriting logic definition.

```
eq stmt(X := A) = aexp(A) -> write(X) .
rl k(int(I) -> write(X) -> K) store(Store)
=> k(K) store(Store[X <- I]) .
```

Ocaml A similar approach as that for Haskell.

```
let rec stmt = function
```

```
    ...
    | Assign(x, a) -> (aexp a) @ [Kwrite x]
let rec result s = function
    ...
    | (Kval (Vint i)::Kwrite x::k) -> result (bind s x i) k
```

Prolog Same approach as for the functional languages above, but we now define (functional) evaluation relations for functions decomposing the program and a one-step rewrite relation for the top-level evaluation process.

```
stmt(X = A,K) :- aexp(A,KA), append(KA,[write(X)],K) .
step(conf(store(S),v([I]),k([write(X)|K])),
     conf(store(S1),v(Vl),k(K)))
     :- bind(S,X,I,S1).
```