# A Rewriting Logic Approach to Operational Semantics*

Traian Florin Şerbănuţă and Grigore Roşu

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

**Abstract**

We show how one can use rewriting logic to *faithfully capture* (not implement) various operational semantic frameworks as rewrite logic theories, namely big-step and small-step semantics, reduction semantics using evaluation contexts, and continuation-based semantics. There is a one-to-one correspondence between an original operational semantics and its associated rewrite logic theory, both notationally and computationally. Once an operational semantics is defined as a rewrite logic theory, one can use standard, off-the-shelf context-insensitive rewrite engines to "execute" programs directly within their semantics; in other words, one gets *interpreters for free* for the defined languages, directly from their semantic definitions. Experiments show that the resulting correct-by-definition interpreters are also reasonably efficient.

## 1 Introduction

Various operational semantics of programming languages have been introduced in the literature, building upon the intuition that executions of programs can be regarded as *context-sensitive reductions*. The context-sensitive nature of operational semantics apparently precludes their execution on standard, context-insensitive term rewriting engines. Typically, one gives a semantics to a language on paper following one or more operational semantics styles, and then, to "execute" it, one implements an interpretor 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 way, in practice there is a gap between the formal operational semantics of the language and its implementation. We argue that such a gap can be eliminated if one uses rewriting logic as a semantic infrastructure for reduction-based operational semantics of languages, in the sense that the semantic definition of a language actually *is* its interpretor. All one needs to do is to execute the language definition on standard existing *context-insensitive term rewrite engines*.

This paper is part of the rewriting logic semantics project [MR06, MR04]. The broad goal of the project is to develop a tool supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on *rewriting logic* [Mes92]. In this paper we focus on only two aspects of the project: (1) the computational relationships between various operational semantics and their corresponding rewrite logic theories, and (2) experiments showing the practical feasibility of our approach. Regarding (1), we show that for a particular language formalized using any of the big-step, small-step or reduction-based using evaluation contexts semantics, say $\mathcal{L}$ , one can devise a rewrite logic theory, say $\mathcal{R}_{\mathcal{L}}$, which is *precisely the intended original language definition $\mathcal{L}$, not an artificial encoding of it*: there is a one-to-one correspondence between derivations using $\mathcal{L}$ and rewrite logic derivations using $\mathcal{R}_{\mathcal{L}}$ (modulo different but minor and ultimately irrelevant notational conventions).

*Supported by NSF grants CCF-0234524, CCF-0448501, CNS-0509321

In a nutshell, rewriting logic is a framework that gives complete semantics (that is, models that make the expected rewrite relation, or "deduction", complete) to the otherwise usual and standard *term rewriting modulo equations*. If one is *not* interested in the model-theoretical semantic dimension of the proposed framework, but only in its operational (including executability, proof theoretical, model checking and, in general, formal verification) aspects, then one can safely think of it as a framework to define programming languages as standard term rewrite systems modulo equations. The semantic counterpart is achieved at no additional cost (neither conceptual nor notational), by just regarding rewrite systems as rewrite logic theories. An immediate advantage of defining a language as a *theory in an existing logic* (as opposed to as a new logic in which one can derive precisely the intended computations), is that one can use the entire arsenal of techniques and tools developed for the underlying logic to obtain corresponding techniques and tools for the particular programming language defined as a theory. For example, the LTL model checker obtained for Java 1.4 from its rewrite logic definition in Maude [CDE+02] compares favorably with state-of-the-art model checkers developed specifically for Java [MR06]. In this paper we also show that rewrite engines capable of executing rewrite logic theories translate into reasonably effective "interpreters" for the defined languages.

The three major contributions of this paper are: (1) a definition of (context-sensitive) reduction semantics using (context-insensitive) rewriting logic; (2) a continuation-based definitional style based on a first-order representation of continuations in rewriting logic, together with its computational equivalence with reduction semantics using evaluation contexts; and (3) experiments showing that the resulting rewrite logic definitions, when executed on rewrite engines, lead to reasonably efficient interpreters. An auxiliary, less major contribution is a small-step operational semantics in rewriting logic in which one can control the number of small-steps to be applied. The methodological and educational contribution is to show, from various perspectives, that rewriting logic can serve as a unifying algebraic framework for operational semantics of programming languages.

# 2 Rewriting Logic

We here informally recall some basic notions of rewriting logic and of its important sublogic called equational logic, together with operational intuitions of term rewriting modulo equations.

***Equational logic*** is perhaps the simplest logic having the full expressivity of computability [BT95]. One can think of it as a logic of "term replacement": terms can be replaced by equal terms in any context. An *equational specification* is a pair $(\Sigma, E)$, where $\Sigma$ is a set of "uninterpreted" operations, also called its "syntax", and $E$ is a set of *equations* of the form $(\forall X)\ t = t'$ constraining the syntax, where $X$ is some set of variables and $t$, $t'$ are well-formed terms over variables in $X$ and operations in $\Sigma$. Equational logics can be *many-sorted* [GM82] (operations in $\Sigma$ have arguments of specific sorts), or even *order-sorted* [GM92], i.e., sorts come with a partial order on them; we use order-sorted specifications in this paper. Also, equations can be *conditional*, where the condition is a (typically finite) set of pairs $u = u'$ over the same variables $X$. We write conditional equations (of finite condition) as $(\forall X)\ t = t' \Leftarrow u_1 = u'_1 \wedge \cdots \wedge u_n = u'_n$. As usual, we drop the quantification in front of the equations and assume universal quantification over all variables occurring in the equation. In equational specifications meant to reason about ground terms, one can add the special condition *otherwise* to an equation, with the meaning that the equation can be used only when other equations have failed at the current position. It is shown in [CDE+02], for example, that this attribute can be easily eliminated by slightly transforming the specification.

***Term rewriting*** is a related approach in which equations are *oriented* left-to-right, written $(\forall X)\ l \to r \Leftarrow u_1 \to u'_1 \wedge \cdots \wedge u_n \to u'_n$ and called *rewrite rules*. It is crucial in our approach that rewrite rules allow *matching in conditions*; in other words, $u'_1$, ..., $u'_n$ can incrementally add new variables to those of $l$, which can be used sequentially in the subsequent conditions and in $r$. A rewrite rule can be applied to a term $t$ at any position where $l$ matches as follows: find some subterm $t'$ of $t$, that is, $t = c[t']$ for some *context* $c$, which is an instance of the left-hand-side term (lhs), that is $t' = \theta_1(l)$ for some variable assignment $\theta_1$, then verify that $\theta_i(u_i) \to^* \theta_{i+1}(u'_i)$ by recursively calling the rewriting engine, where $\theta_{i+1}$ extends $\theta_i$ with substitutions for the variables in $u'_i$ but not bound yet by $\theta_i$; when all conditions are satisfied, replace $t'$ by $\theta_n(r)$ in $t$. This way, the term $t$ can be continuously transformed, or rewritten. A pair $(\Sigma, R)$, where $R$ is a set of such

oriented *rewrite rules*, is called a *rewrite system*. The corresponding term rewriting relation is written $\rightarrow_R$. If no rule in $R$ can rewrite a $\Sigma$-term, than that term is said to be in *normal form* w.r.t. $R$.

Term rewriting can be used as an operational mechanism to perform equational deduction; we say that equational specifications can be executed by rewriting. There are many software systems that either specifically implement term rewriting efficiently, known also 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 [vdBHKO02], CafeOBJ [DF98], Elan [BCD$^+$00], Maude [CDE$^+$02], OBJ [GWM$^+$93], Stratego [Vis03]. Some of these can achieve remarkable speeds on today's machines, in the order of tens of millions of rewrite steps per second.

Because of the forward chaining executability of term rewriting and also because of these efficient rewrite engines, equational specifications are often called *executable*. As programming languages tend to be increasingly more abstract due to the higher speeds of processors, and as specification languages tend to be provided with faster execution engines, the gap between executable specifications and implementations, in case there has ever been any, is becoming visibly narrower, that is, the pragmatic, semantics-reluctant language designer, can safely regard the subsequent semantic definitions of language features as implementations, in spite of their conciseness and mathematical flavor.

While equational logic and its execution via term rewriting provide as powerful computational properties as one can get in a sequential setting, these were *not* designed to specify or reason about *transitions*. The initial algebra model of an equational specification collapses all the computationally equivalent terms, but it does not say anything about *evolution* of terms.

***Rewriting logic*** [Mes92], which should not be confused with term rewriting, is a logic for concurrency. A *rewrite theory* is a triple $(\Sigma, E, R)$, where $(\Sigma, E)$ is an equational specification and $R$ is a set of *rewrite rules*. Rewriting logic therefore extends equational logic with rewrite rules, allowing one to derive both equations and rewrites (or transitions). Deduction remains the same for equations, but the symmetry rule is dropped for rewrite rules. Models of rewrite theories are $(\Sigma, E)$-algebras enriched with transitions satisfying all the rewrite rules in $R$. The distinction between equations and rewrite rules is only semantic. They are both executed as rewrite rules by rewrite engines, following the simple, uniform and parallelizable principle of term rewriting. Therefore, if one is interested in just a dynamic semantics of a language, one needs to make no distinction between equations and rewrite rules. Rewriting logic is a framework for *true concurrency*: the locality of rules, given by their context-insensitiveness, allows multiple rules to apply at the same time provided they don't modify the shared part.

## 2.1 Rewriting Logic Deduction

Given $\mathcal{R} = (\Sigma, E \cup A, R)$, 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**. $\dfrac{(\forall X)\ u \longrightarrow v \quad E \cup A \vdash (\forall X) u = u' \quad E \cup A \vdash (\forall X) v = v'}{(\forall X)\ u' \longrightarrow v'}$
- **Congruence**. For each $f : s_1 \dots s_n \longrightarrow s$ in $\Sigma$, with $t_i \in T_\Sigma(X)_{s_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{s_{j_l}}$, $1 \leq l \leq m$,

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

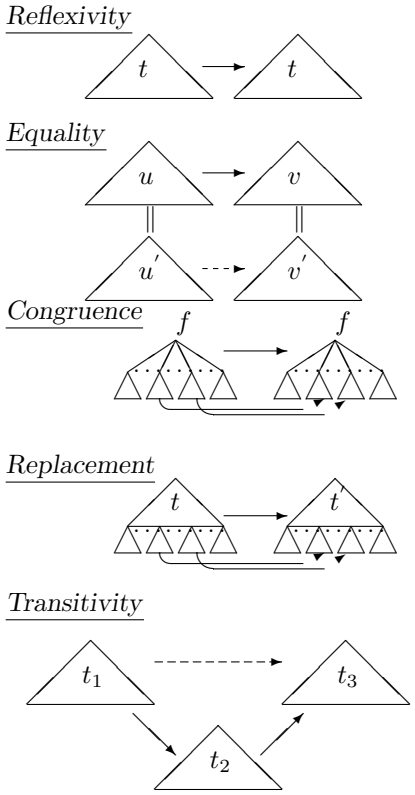


Figure 1: Visual representation of rewriting logic deduction.

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

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

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

- **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 nonfrozen 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 lefthand side to the corresponding righthand side instance, the state fragments in the substitution

of the rule's variables can also be rewritten, provided the variables involved are not frozen. Finally, the "Transitivity" rule allows us to build longer concurrent computations by composing them sequentially.

When doing proofs, is it usually easier to work with a "one step" relation $\rightarrow^1$, defined on ground terms. By abuse of notation, we say that $\mathcal{R} \vdash t \rightarrow^1 t'$ iff $t$ and $t'$ satisfy the following:

1. $\mathcal{R} \vdash t \rightarrow t'$;
2. For any $t''$ such that $\mathcal{R} \vdash t \rightarrow t''$ and $\mathcal{R} \vdash t'' \rightarrow t'$ we have that $E \cup A \vdash t'' = t$ or $E \cup A \vdash t'' = t'$; and
3. $E \cup A \not\vdash t = t''$.

The relation $\rightarrow^{\leq 1}$ is defined requiring all but the last condition. Similarly, one can define relations $\rightarrow^n$ (or $\rightarrow^{\leq n}$) by requiring the existence of terms $(t_i)_{0 \leq i \leq n}$ such that $E \cup A \vdash t_0 = t$, $E \cup A \vdash t_n = t'$ and for $0 \leq i < n$, $\mathcal{R} \vdash t_i \rightarrow^1 t_{i+1}$ (or $\mathcal{R} \vdash t_i \rightarrow^{\leq 1} t_{i+1}$). Note that, the above convention also defines $\rightarrow^0$ as the pairs of terms provable without using rules.

We define programming languages as rewrite logic theories. Specifically, in this approach, we use the fact that rewriting logic deduction is done modulo equations to faithfully capture the computational granularity of a language by making rewriting rules all intended computational steps, while using equations for convenient equivalent structural transformations of the state which should not be regarded as computation steps. Since rewriting logic is a *computational logical framework*, "execution" of programs becomes logical deduction. That means that one can formally analyze programs or their executions directly within the semantic definition of their programming language. In particular, executions can be regarded as proofs, so one can log and check them, thus obtaining a framework for *certifiable execution* of programs. Moreover, generic analysis tools for rewrite logic specifications can translate into analysis tools for the defined programming languages. For example, Maude provides a BFS reachability analyzer and an LTL model checker for rewrite logic theories; these translate immediately into corresponding BFS reachability analysis and LTL model checking tools for the defined languages, also *for free*. In this paper we only stress the capability of rewrite engines to execute rewrite logic theories, thus yielding language interpreters. All our definitions are interpreted by rewrite engines as *deterministic conditional rewrite systems* for execution purposes. A conditional term rewrite system is *deterministic* if any rule $r_0 \rightarrow l_{n+1} \Leftarrow l_1 \rightarrow r_1 \wedge \ldots \wedge l_n \rightarrow r_n$ satisfies $vars(l_j) \subseteq \bigcup_{k=0}^{j-1} vars(r_k)$. A rewrite engine supporting deterministic conditional term rewrite systems solves conditions from left to right, accumulating the substitution. Most currently available rewriting engines support this very general form of rewriting. We will also make use of it when presenting various semantics as rewriting logic theories.

# 3 A Simple Imperative Language

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

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

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

This BNF syntax is entirely equivalent with an algebraic 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 operation

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

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

The next sections will present rewriting logic definitions of various operational semantics ( big step, small step and reduction using evaluation contexts styles) for this simple language as well as their corresponding standard operational semantics definitions, stating the relation between them and pointing a set of strengths and weaknesses for each framework. The reader is referred to [Kah87, Plo04, WF94] for further details on the described operational semantics.

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

# 4 Store

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

To define the store, we assume a pairing "binding" constructor "$\_ \mapsto \_$", associating values to variables, and an associative and commutative operation "$\_\_$" of unit $\emptyset$ to put together such bindings. Equational definitions $\mathcal{E}_{\mathit{Store}}$ of operations $\_[\_]$ to retrieve the value of a variable in the store and $\_[\_ \leftarrow \_]$ to update the value of a variable are:

$$\begin{array}{l}
(S\ X \mapsto I)[X] = I \\
(S\ X \mapsto I)[X'] = S[X'] \Leftarrow X \neq X' \\
(S\ X \mapsto I)[X \leftarrow I'] = S\ X \mapsto I' \\
(S\ X \mapsto I)[X' \leftarrow I'] = S[X' \leftarrow I']\ X \mapsto I \Leftarrow X \neq X' \\
\emptyset[X \leftarrow I] = X \mapsto I
\end{array}$$

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

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

**Proposition 1** *Let $x, x' \in \mathit{Var}$, $i, i' \in \mathit{Int}$, $s, s' \in \mathit{Store}$ and $\sigma, \sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$.*

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

*Proof.*

1. Trivial, since $\mathcal{E}_{\mathit{Store}} \nvdash \emptyset[x] = i$ for any $x \in \mathit{Var}, i \in \mathit{Int}$.

2. Consider an arbitrary $x'$. If $x' = x$, then $\mathcal{E}_{Store} \not\vdash s[x'] = i'$ for any $i'$, since otherwise we would have $\mathcal{E}_{Store} \vdash s = s'\ x \mapsto i'$ which contradicts the well definedness of $s\ x \mapsto i$. If $x' \neq x$, then $\mathcal{E}_{Store} \vdash s[x'] = (s\ x \mapsto i)[x']$.

3. Suppose $s \simeq \sigma$. We distinguish two cases - if $\sigma$ is defined on $x$ or if it is not. If it is, then let us say that $\sigma(x) = i'$; in that case we must have that $\mathcal{E}_{Store} \vdash s[x] = i'$ 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 $\sigma$ is not defined for $x$, it means that $\mathcal{E}_{Store} \not\vdash s[x] = i$ for any $i$, whence $\mathcal{E}_{Store} \not\vdash s = s'\ x \mapsto i$. If $\mathcal{E}_{Store} \vdash s = \emptyset$ then we are done, since $\mathcal{E}_{Store} \vdash (x \mapsto i)[x'] = i'$ iff $x = x'$ and $i = i'$. If $\mathcal{E}_{Store} \not\vdash s = \emptyset$, it must be that $\mathcal{E}_{Store} \vdash s = x_1 \mapsto i_1 \ldots x_n \mapsto i_n$ with $x_i \neq x$. This leads to $\mathcal{E}_{Store} \vdash s[x \leftarrow i] = \cdots = (x_1 \mapsto i_1 \ldots x_i \mapsto i_i)[x \leftarrow i](x_{i+1} \mapsto i_{i+1} \ldots x_n \mapsto i_n) = \cdots = \emptyset[x \leftarrow i]s = (x \mapsto i)s = s(x \mapsto i)$.

□

# 5 Big-Step Operational Semantics

Introduced as natural semantics in [Kah87], also named relational semantics [MTHM97] 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:

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

The complete[1] big-step operational semantics definition for our simple language except its `halt` statement (which is discussed at the end of this section), say *BigStep*, is presented in Table 1. To give a rewriting logic theory for the big-step semantics, above, 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 rewriting logic theory $\mathcal{R}_{BigStep}$ corresponding to the big-step operational semantics *BigStep* entirely automatically as shown by Table 2.

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

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

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

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

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

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

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

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

---

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

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

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

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

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

---

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

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

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

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

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

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

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

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

---

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

Table 1: The *BigStep* language definition

$$
\begin{array}{rcl}
\langle X, S\rangle \to \langle S[X], S\rangle & & \\
\langle \#I, S\rangle \to \langle I, S\rangle & & \\
\langle \texttt{++}X, S\rangle \to \langle I, S[X \leftarrow I]\rangle & \Leftarrow & I = S[X] + 1 \\
\langle A_1 + A_2, S\rangle \to \langle I_1 +_{Int} I_2, S_2\rangle & \Leftarrow & \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 \\
\hline
\langle \#T, S\rangle \to \langle T, S\rangle & & \\
\langle A_1 \texttt{<=} A_2, S\rangle \to \langle (I_1 \leq_{Int} I_2), S_2\rangle & \Leftarrow & \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 & \Leftarrow & \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 & \Leftarrow & \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 & \Leftarrow & \langle B, S\rangle \to \langle T, S'\rangle \\
\hline
\langle \mathit{skip}, S\rangle \to \langle S\rangle & & \\
\langle X \texttt{:=} A, S\rangle \to \langle S'[X \leftarrow I]\rangle & \Leftarrow & \langle A, S\rangle \to \langle I, S'\rangle \\
\langle St_1; St_2, S\rangle \to \langle S'\rangle & \Leftarrow & \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 & \Leftarrow & \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 & \Leftarrow & \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 & \Leftarrow & \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 \texttt{ } St, S\rangle \to \langle S'\rangle & \Leftarrow & \langle B, S\rangle \to \langle \mathit{false}, S'\rangle \\
\langle \texttt{while } B \texttt{ } St, S\rangle \to \langle S'\rangle & \Leftarrow & \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 \texttt{ } St, S_2\rangle \to \langle S'\rangle \\
\hline
\langle St.A\rangle \to \langle I\rangle & \Leftarrow & \langle St, \emptyset\rangle \to \langle S\rangle \wedge \langle A, S\rangle \to \langle I, S'\rangle
\end{array}
$$

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

*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 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 ineffective. Suppose $s \in \mathit{Store}$ and $\sigma : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ such that $s \simeq \sigma$. We prove the following affirmations:

1. $\mathit{BigStep} \vdash \langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ iff $\mathcal{R}_{BigStep} \vdash \langle a, s\rangle \to \langle i, s'\rangle$ 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 $\mathcal{R}_{BigStep} \vdash \langle b, s\rangle \to \langle t, s'\rangle$ 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 $\mathcal{R}_{BigStep} \vdash \langle st, s\rangle \to \langle s'\rangle$ 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 $\mathcal{R}_{BigStep} \vdash \langle p\rangle \to \langle i\rangle$,
   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 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
   $\mathcal{E}_{Store} \subseteq \mathcal{R}_{BigStep} \vdash i = s[x] + 1$ iff
   $\mathcal{R}_{BigStep} \vdash \langle x\texttt{++}, s\rangle \to \langle i, s[x \leftarrow i]\rangle$.
   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
   ($\mathcal{R}_{BigStep} \vdash \langle b_1, s\rangle \to \langle \mathit{false}, s'\rangle$, $s' \simeq \sigma'$ and $t = \mathit{false}$
   or $\mathcal{R}_{BigStep} \vdash \langle b_1, s\rangle \to \langle \mathit{true}, s''\rangle$, $s'' \simeq \sigma''$, $\mathcal{R}_{BigStep} \vdash \langle b_2, s''\rangle \to \langle t, \sigma'\rangle$ and $s' \simeq \sigma'$ ) iff
   $\mathcal{R}_{BigStep} \vdash \langle b_1 \texttt{ and } b_2, s\rangle \to \langle t, s'\rangle$ and $s' \simeq \sigma'$.

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

This completes the proof. □

The only apparent difference between *BigStep* and $\mathcal{R}_{BigStep}$ is the different notational conventions they use. However, 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 $\mathcal{R}_{BigStep}$, in order to be faithfully equivalent to *BigStep* computationally, 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 to apply only at the top, sequentially).

***Strengths*** of big-step operational semantics: straight-forward recursive definition; when deterministic, can be easily and efficiently interpreted in any recursive, functional or logical framework; specifically useful for defining type systems.

***Weaknesses*** of big-step operational semantics are due to its monolithic, single-step evaluation: it is hard to debug or trace; if the program is wrong, no information is given about where the failure occurred; it is hard or impossible to model concurrent features; it is not modular - 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 adding halt to the above definition - one needs to add a special configuration $halting(I)$, and the following rules:

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

# 6 Small-Step Operational Semantics

Introduced in [Plo04], also called transition semantics or reduction semantics, small-step semantics captures the notion of one computational step.

One inherent complication in capturing small-step operational semantics as a rewriting logic theory in a one-to-one notational and computational correspondence is that the rewriting relation is by definition transitive, while the small-step relation is *not* transitive (its transitive closure is defined aposteriori). Therefore we need to devise a mechanism to "inhibit" rewriting logic's transitive and uncontrolled application of rules.

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

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

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

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

---

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

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

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

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

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

---

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

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

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

---

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

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

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

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

---

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

---

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

Table 3: The *SmallStep* language definition

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

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

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.,

$$\cdot\langle A_1 + A_2, S\rangle \to \langle A_1' + A_2, S'\rangle \Leftarrow \cdot\langle A_1, S\rangle \to \langle A_1', S'\rangle$$

A similar technique is proposed in [MB04], 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 part of 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[2] small-step operational semantics definition for our simple language except its `halt` statement (which is discussed at the end of this section), say *SmallStep*, is presented in Table 3. The corresponding small-step rewriting logic theory $\mathcal{R}_{\mathit{SmallStep}}$ is given in Table 4

As for big-step, the rewriting under context deduction rule for rewriting logic is again ineffective, since all rules act at the top, on configurations. However, it is not the case in *SmallStep* that all right hand sides are normal forms (this actually is the specificity of small-step semantics). The "$\cdot$" operator introduced in $\mathcal{R}_{\mathit{SmallStep}}$ 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 $p \in \mathit{Pgm}$, $\sigma, \sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ and $s \in \mathit{Store}$ such that $s \simeq \sigma$, the following are equivalent:*

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

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

1. $\mathit{SmallStep} \vdash \langle p, \bot\rangle \to^* \langle \mathtt{skip}.i, \sigma\rangle$ *for some* $\sigma : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$, *and*
2. $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{eval}(p) \to i$.

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

1. $\mathit{SmallStep} \vdash \langle a, \sigma\rangle \to \langle a', \sigma'\rangle$ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \cdot\langle a, s\rangle \to \langle a', s'\rangle$ and $s' \simeq \sigma'$, for any $a, a' \in \mathit{AExp}$, $\sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ and $s' \in \mathit{Store}$.
2. $\mathit{SmallStep} \vdash \langle b, \sigma\rangle \to \langle b', \sigma'\rangle$ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \cdot\langle b, s\rangle \to \langle b', s'\rangle$ and $s' \simeq \sigma'$, for any $b, b' \in \mathit{BExp}$, $\sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ and $s' \in \mathit{Store}$.
3. $\mathit{SmallStep} \vdash \langle st, \sigma\rangle \to \langle st', \sigma'\rangle$ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \cdot\langle st, s\rangle \to \langle st', s'\rangle$ and $s' \simeq \sigma'$, for any $st, st' \in \mathit{Stmt}$, $\sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ and $s' \in \mathit{Store}$.
4. $\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \cdot\langle p, s\rangle \to \langle p', s'\rangle$ and $s' \simeq \sigma'$, for any $p, p' \in \mathit{Pgm}$, $\sigma' : \mathit{Var} \xrightarrow{\circ} \mathit{Int}$ and $s' \in \mathit{Store}$.

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

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

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

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

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
   $R_{\mathit{SmallStep}} \vdash \cdot\langle st_1, s\rangle \to \langle st_1', s'\rangle$ and $s' \simeq \sigma'$ iff
   $R_{\mathit{SmallStep}} \vdash \cdot\langle st_1; st_2, s\rangle \to \langle st_1' + st_2, s'\rangle$ 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
   $st = \texttt{skip}$, $R_{\mathit{SmallStep}} \vdash \cdot\langle a, s\rangle \to \langle a', s'\rangle$ 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 pair which can be provable by exactly applying $n-1$ transitivity rules for $\to$ if $n > 0$ or the diagonal 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 $s \simeq \sigma$):

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

by induction on $n$. If $n = 0$ then $\langle p, \sigma\rangle = \langle p', \sigma'\rangle$ and since $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) = \mathit{smallstep}(\cdot\langle p, s\rangle)$ we are done. If $n > 0$, we have that
$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to^n \langle p', \sigma'\rangle$ 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 $s' \simeq \sigma'$ (by the induction hypothesis) iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle p, s\rangle) \to^1 \mathit{smallstep}(\langle p_1, s_1\rangle)$ 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*** of small-step operational semantics: precisely defines the notion of one computational step; stops at errors, pointing them out; easy to trace, debug; gives interleaving semantics for concurrency.

***Weaknesses*** of small-step operational semantics: it is an "iterated big step", that is, each small step does the same amount of computation as a big step in finding the next redex; does not give a "true concurrency" semantics, that is, one has to chose a certain interleaving (no two rules can apply at the same moment), 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 is not entirely accurate: for example, one needs to have in the syntax boolean values if one want to have boolean expressions and needs an `if` mechanism in the above definition to evaluate `while` - the fact that this are

common in programming languages does not mean that the languages which don't want to allow them should be despised; still hard to deal with control - for example, consider adding halt to this language. One cannot simply do it as for other ordinary statements - instead one has to add a corner case (additional rule) to each statement, as shown below:

$$\begin{array}{r} \cdot\langle \texttt{halt}\ A, S\rangle \to \langle \texttt{halt}\ A', S'\rangle \\ \cdot\langle \texttt{halt}\ I; St, S\rangle \to \langle \texttt{halt}\ I, S\rangle \\ \cdot\langle \texttt{halt}\ I.A, S\rangle \to \langle \texttt{skip}.I, S\rangle \end{array} \quad \Leftarrow \quad \cdot\langle A, S\rangle \to \langle A', S'\rangle$$

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

# 7 Reduction Semantics with Evaluation Contexts

Introduced in [WF94], also called context reduction, the evaluation contexts style improves over small-step definitional style in two directions:

1. 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 gives one the possibility to also modify 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, $C[\texttt{halt}\ I] \to I$, preserving the desired computation granularity.

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

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

capturing the fact that reductions are allowed to take place only in appropriate evaluation contexts. Therefore, an important part of a context reduction semantics is the definition of evaluation contexts, which is typically done by means of a context-free grammar. The definition of evaluation contexts for our simple language is found in Table 5 (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 where that is not needed, but instead one needs to employ some substitution mechanism (particularly in definitions of $\lambda$-calculus based languages). The rules following the contexts grammar in Table 5 complete the context reduction semantics of our simple language, say *CxtRed*.

By making the evaluation context explicit and changeable, context reduction is, in our view, a significant improvement over small-step SOS. In particular, one can now define control-intensive statements like `halt` *modularly* and at the desired computational granularity level. Even though the definition in Table 5 gives one the feel that evaluation contexts and their instantiation come "for free", the application of the "rewrite in context" rule presented above can be expensive in practice. That 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

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

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

---

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

---

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

---

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

---

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

Table 5: The *CxtRed* language definition

context reduction such as `PLT-Redex` cannot avoid paying a significant performance penalty, as the numbers in Table 12 show.

Context reduction is trickier to faithfully capture as a rewriting logic theory, since rewriting logic, by its locality, always applies a rule *in* the context, without actually having the capability to change the context. 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. *s2c*, presented in Table 6 has an effect similar to what one achieves by parsing in context reduction, in the sense that given a piece of syntax it yields $C[R]$. *c2s*, also presented in Table 6 is defined as a morphism on the syntax, but we get (from the rules) the guarantee that it would be applied only to "well-formed" contexts (i.e., containing only one hole). The rewriting logic theory $\mathcal{R}_{CxtRed}$ is obtained by adding the rules in Table 7 to the equations of *s2c* and *c2s*.

The $\mathcal{R}_{CxtRed}$ definition is a faithful representation of reduction semantics using evaluation contexts: 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 the same way as other current implementations of this technique do it. Also since parsing issues are abstracted away using equations, computational granularity is the same, having a one-to-one correspondence between the reduction semantics rules and the rewriting rules.

**Theorem 1** *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 $\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s\rangle) \to \langle p', s'\rangle$ and $s' \simeq \sigma'$;*
4. *$CxtRed \vdash \langle p, \sigma\rangle \to \langle i\rangle$ iff $\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s\rangle) \to \langle i\rangle$;*
5. *$CxtRed \vdash \langle p, \bot\rangle \to^* \langle i\rangle$ iff $\mathcal{R}_{CxtRed} \vdash eval(p) \to i$.*

*Proof.*

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

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

   **Case** $a_1$`<=`$a_2$**:** $a_1$ `<=` $a_2$ 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** $x$`:=`$a$**:** $x$`:=`$a$ 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
   $\mathcal{R}_{CxtRed} \vdash s2c(\langle p, s\rangle) = \langle c', s\rangle[r]$.

$$s2c(\langle P, S\rangle) = \langle C, S\rangle[R] \Leftarrow C[R] = s2c(P)$$

---

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

---

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

---

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

---

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

$$c2s([][H]) = H$$

---

$$c2s(\langle P, S\rangle[H]) = \langle c2s(P[H]), S\rangle$$
$$c2s(\langle I\rangle[H]) = \langle I\rangle$$

---

$$c2s(E_1.E_2[H]) = c2s(E_1[H]).c2s(E_2[H])$$

---

$$c2s(\texttt{halt}\ E[H]) = \texttt{halt}\ c2s(E[H])$$
$$c2s(\texttt{while}\ E_1\ \ E_2[H]) = \texttt{while}\ c2s(E_1[H])\ \ c2s(E_2[H])$$
$$c2s(\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2[H]) = \texttt{if}\ c2s(E[H])\ \texttt{then}\ c2s(E_1[H])\ \texttt{else}\ c2s(E_2[H])$$
$$c2s(\{E\}[H]) = \{c2s(E[H])\}$$
$$c2s(E_1; E_2[H]) = c2s(E_1[H]); c2s(E_2[H])$$
$$c2s(X\texttt{:=}E[H]) = X\texttt{:=}c2s(E[H])$$
$$c2s(\texttt{skip}[H]) = \texttt{skip}$$

---

$$c2s(E_1\texttt{<=}E_2[H]) = c2s(E_1[H])\texttt{<=}c2s(E_2[H])$$
$$c2s(E_1\ \texttt{and}\ E_2[H]) = c2s(E_1[H])\ \texttt{and}\ c2s(E_2[H])$$
$$c2s(\texttt{not}\ E[H]) = \texttt{not}\ c2s(E[H])$$
$$c2s(\texttt{true}[H]) = \texttt{true}$$
$$c2s(\texttt{false}[H]) = \texttt{false}$$

---

$$c2s(\texttt{++}X[H]) = \texttt{++}X$$
$$c2s(E_1 + E_2[H]) = c2s(E_1[H]) + c2s(E_2[H])$$
$$c2s(I[H]) = I$$

Table 6: Equational definitions of *s2c* and *c2s*

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

Table 7: $\mathcal{R}_{CxtRed}$ rewriting logic theory

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., contexts with multiple holes), but that aspect does not interest us here.

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

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

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

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

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

$\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s\rangle) = \langle i\rangle$, since $\mathcal{R}_{CxtRed} \vdash \cdot(\langle [], \sigma\rangle[\texttt{skip}.i]) \rightarrow \langle i\rangle[[]]$ and since $\mathcal{R}_{CxtRed} \vdash c2s(\langle i\rangle[[]]) = \langle i\rangle$.

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

5. This proof follows exactly like the one for the similar property for *SmallStep* (Proposition 3), using the above properties and replacing *smallstep* by *reduction*.

□

***Strengths*** of context reduction semantics: distinguishes small-step rules into computational rules and rules needed to find the redex (the latter are transformed into grammar rules generating the allowable contexts), making definitions more compact; it improves over small step by allowing the context to be changed by execution rules; can deal easily with control intensive features; more modular than SOS.

***Weaknesses*** of context reduction semantics: still allows only "interleaving semantics" for concurrency; although it would suggest context-sensitive rewriting, all current implementations 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).

# 8 A Continuation Based Semantics in Rewriting Logic

The idea of continuation-based interpretors for programming languages and their relation to abstract machines was well studied (see, for example, [FF86]). In this section we propose a rewrite logic theory based on a structure that appears to be a *first-order* representation of continuations; that is the only reason for which we call it "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 [FF86] to understand this section. We will show the equivalence of this theory to the context reduction semantics theory.

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

The 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 more obvious: equations are used to prepare the context in which a computation step can be applied, while rewrite rules exactly encode the computation steps semantically, yielding the intended 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 identity a constant "*nothing*". We also use lists of values and continuations, each having an associative 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 $\mathcal{R}_K$ rewriting logic continuation-based definition of our language is given in Table 8.

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

---

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

---

$stmt(\texttt{skip}) = nothing$
$stmt(X := A) = aexp(A) \curvearrowright write(X)$
$stmt(St_1; St_2) = stmt(St_1) \curvearrowright stmt(St_2)$
$stmt(\{St\}) = stmt(St)$
$stmt(\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2) = bexp(B) \curvearrowright if(stmt(St_1), stmt(St_2))$
$stmt(\texttt{while}\ B\ St) = bexp(B) \curvearrowright while(bexp(B), stmt(St))$
$stmt(\texttt{halt}\ A) = aexp(A) \curvearrowright halt$
$k(I \curvearrowright write(X) \curvearrowright K)\ store(Store) \rightarrow k(K)\ store(Store[X \leftarrow I])$
$k(true \curvearrowright if(K_1, K_2) \curvearrowright K) \rightarrow k(K_1 \curvearrowright K)$
$k(false \curvearrowright if(K_1, K_2) \curvearrowright K) \rightarrow k(K_2 \curvearrowright K)$
$k(true \curvearrowright while(K_1, K_2) \curvearrowright K) \rightarrow k(K_2 \curvearrowright K_1 \curvearrowright while(K_1, K_2) \curvearrowright K)$
$k(false \curvearrowright while(K_1, K_2) \curvearrowright K) \rightarrow k(K)$
$k(I \curvearrowright halt \curvearrowright K) \rightarrow k(I)$

---

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

---

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

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

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

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

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

Table 8: $\mathcal{R}_K$ rewriting logic theory (continuation-based definition of the language)

The most important benefit of this transformation is that of gaining locality. Now one needs to specify from the context only what is needed to perform the computation. This indeed gives the possibility of achieving "true concurrency", since rules which do not act on same parts of the context can be applied in parallel. We here only discuss the sequential variant of our continuation-based semantics, because our language is sequential. In [Roş05] we show how the same technique can be used at 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*** of continuation based semantics: no need to search for a redex anymore, the redex is always at the top; much more efficient than *direct* implementations of evaluation contexts or small-step SOS.

> ***Weaknesses*** of continuation based semantics: 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 get programs from continuations[3]; to flatten the program into a continuation structure, several new operations (continuation constants) need to be introduced, which "replace" the corresponding original language constructs.

An important "strength" specific to the rewriting logic approach is that reductions can now apply wherever they match, *context-insensitively*. Also, this style eliminates the need for conditional rules, which might involve reachability analysis to check the conditions and are harder to deal with in parallel environments.

## 8.1 Proofs

An important aspect of formal definitions of languages, in contrast to just implementing interpretors, is that they allow formal proofs to be done. In this section we show how one could prove equivalences of programs within a rewriting logic theory defining a programming language. Specifically, we here prove the soundness of loop unrolling. But first, we formally define what equivalence of programs means in this framework. Given a rewriting logic theory $\mathcal{R} = (\Sigma, E, R)$ and a (ground) $\Sigma$-term $t$, let $\lfloor t \rfloor_E$ denote the class of (ground) $\Sigma$-terms provably equal to $t$, i.e., $\lfloor t \rfloor_E = \{t' \in T_\Sigma \mid E \vdash t = t'\}$. For a set of $\Sigma$-equations $E'$, the *$E'$-congruence* is the relation containing all pairs of $\Sigma$-terms that can be proved equal using $E'$.

**Definition 1** *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewriting logic theory with a distinguished visible equational subtheory $(\Sigma_V, E_V)$ of* value datatypes *(or, the* data universe*) which is protected by $\mathcal{R}$ in the sense that for any $\Sigma_V$-terms $v_1, v_2$, if $\mathcal{R} \vdash v_1 \to v_2$ then[4] $E_V \vdash v_1 = v_2$; for this reason, we call $\Sigma_V$ terms $v_1$, $v_2$, etc., as well as the $\Sigma$-terms in their $E$-equivalence classes, $\lfloor v_1 \rfloor_E$, $\lfloor v_2 \rfloor_E$, etc.,* visible.

*A* computational congruence $\approx$ *on $\mathcal{R}$ is a congruence on $T_\Sigma$ such that:*

- *it includes the $E$-congruence;*
- *on $\Sigma_V$-terms, it is precisely the $E_V$-congruence;*
- *it preserves the computational granularity of $\mathcal{R}$, i.e., if $t_1 \approx t_2$ then there exists a bijection associating to each $\lfloor t_1' \rfloor_E$ such that $\mathcal{R} \vdash t_1 \to^1 t_1'$ an $E$-equivalence class $\lfloor t_2' \rfloor_E$ such that $\mathcal{R} \vdash t_2 \to^1 t_2'$ and $t_1' \approx t_2'$.*

*A* functional congruence *is obtained if we drop the last requirement, of preservation of computational granularity.*

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

[4] Since in rewriting logic the relation "$\to$" is reflexive and is defined on top of equational deduction, "$\mathcal{R} \vdash v_1 \to v_2$" also includes the case when $\lfloor v_1 \rfloor_E = \lfloor v_2 \rfloor_E$.

There is some similarity between our notion of computational granularity preservation and the notion of bisimulation [Mil89]. However, note that our notion of computational congruence above differs from the notion of bisimulation on $E$-equivalence classes, in that we require it to "protect" the data universe, or the values (second bullet), while in bisimulation based approaches the data tends to be encoded as labels of transitions. Rewriting logic also allows labeled transitions [Mes92], but we chose not to consider them yet in our definitions.

Notice that the $E$-congruence is a computational congruence, more precisely the *minimal* one. We will show that there also exists a *maximal* such congruence, that we call *the computational equivalence*. But first, let us define the notion of a trace tree.

**Definition 2** *For each "invisible" term $t$ (i.e., $t$ is not $E$-equivalent to any visible term), let $Next(t)$ be the set of "states" (i.e., $E$-equivalence classes) reachable in one step from $t$: $Next(t) = \{\lfloor t' \rfloor_E \mid \mathcal{R} \vdash t \rightarrow^1 t'\}$. Let the* trace tree *of a "state", $Tree(\lfloor t \rfloor_E)$, be the tree rooted in $\cdot$ and having as direct descendants the subtrees $Tree(\lfloor t' \rfloor_E)$ for all distinct elements $\lfloor t' \rfloor_E \in Next(t)$. For any visible term $v$, $Tree(\lfloor v \rfloor_E)$ is the tree $\cdot \stackrel{\lfloor v \rfloor_E}{\longrightarrow} \cdot$.*

Therefore, our trace trees contain labels only on the edges leading to leaves. The other edges contain no labels because, for the time being, we prefer to make no distinction among the various types of computational steps: all it matters is their number and the resulting values. For deterministic programs, trace trees will be just plain linear traces. Proper trees can result when non-deterministic languages are defined. We therefore prefer a tree-based, rather than a trace-based, equivalence in the context of non-determinism. We next define the important notion of computational equivalence as a contextual "behavioral" equivalence:

**Definition 3** *The* computational equivalence $\equiv_{\mathcal{R}}$ *on $\mathcal{R}$ is defined by:*

$$t_1 \equiv_{\mathcal{R}} t_2 \text{ iff for any context } c, \ Tree(\lfloor c[t_1] \rfloor_E) \text{ and } Tree(\lfloor c[t_2] \rfloor_E) \text{ are bisimilar.}$$

**Theorem 2** *For any $\mathcal{R}$ as in Definition 1, the computational equivalence is the greatest computational congruence.*

*Proof.* Fist, let us prove that $\equiv_{\mathcal{R}}$ is a computational congruence in the sense of Definition 1. That it is an equivalence is clear because the bisimulation relation is an equivalence; the compatibility with operations follows by transitivity and the fact that we can put any context on top of a term. $\equiv_{\mathcal{R}}$ includes the $E$-congruence because its definition refers only to classes of $E$-equivalent terms. $\equiv_{\mathcal{R}}$ is the $E_V$ equivalence on visible terms since $v_1 \equiv_{\mathcal{R}} v_2$ implies in particular that $Tree(v_1)$ is bisimilar to $Tree(v_2)$, so they must be labeled by the same $E$-equivalence class. The computational granularity condition follows from the similar condition in the definition of bisimulation of trees [Mil89].

Let us now show that any other computational congruence $\approx$ is smaller that $\equiv_{\mathcal{R}}$. From the congruence condition one can derive that $c[t_1] \approx c[t_2]$ for any $t_1 \approx t_2$. The bisimilarity condition follows by induction on the structure of trace trees, using the preservation of computational granularity property for $\approx$. □

**Corollary 1** *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewriting logic theory as in Definition 1, and let $t_1$, $t_2$ be two ground $\Sigma$-terms such that the $(E \cup \{t_1 = t_2\})$-congruence is a computational congruence on $\mathcal{R}$. Then $t_1 \equiv_{\mathcal{R}} t_2$.*

*Proof.* Obvious, since the pair $(t_1, t_2)$ is in the $(E \cup \{t_1 = t_2\})$-congruence which, as a computational congruence is included in the computational equivalence (by Theorem 2). □

The above gives us a generic way to prove that two terms $t_1$ and $t_2$ are computationally equivalent: prove that the $E \cup \{t_1 = t_2\}$-congruence is a computational congruence on $\mathcal{R}$. Since the $(E \cup \{t_1 = t_2\})$-congruence includes the $E$-congruence (first item in Definition 1), one only needs to show that the new equation does not affect the data universe (second item in Definition 1) and that it still preserves the computational granularity of $\mathcal{R}$. Regarding the former issue, first note that one can indeed destroy the data consistency by adding inappropriate equalities $t_1 = t_2$; for example, if one adds $aexp(7) = aexp(5)$ then one can derive $7 = 5$, which

is obviously not desirable within the original $(\Sigma_V, E_V)$. The data consistency preservation problem can be made arbitrarily complex in worst case scenarios; however, as we shall shortly see, in practice one can show it relatively easily. Regarding the preservation of computational granularity, one can show it by analyzing the effect that each rule has on contexts containing the two terms.

**Proposition 4** *In the $\mathcal{R}_K$ language definition in Table 8, suppose that the data universe is given by the assumed signatures and equations for integers and booleans. Then for any $b \in BExp$ and $st \in Stmt$,*

$$\texttt{while } b\ st \equiv_{\mathcal{R}_K} \texttt{if } b \texttt{ then } \{st; \texttt{while } b\ st\} \texttt{ else skip}$$

*Proof.* Since this is the first proof of computational equivalence of programs in this setting, we take the risk of being boring and prefer to discuss in depth the details of the proof. Since the two terms are in normal form with regard to $\mathcal{R}$ and since they can only be placed in a $stmt(\cdot)$ context, it suffices to prove that

$$stmt(\texttt{while } b\ st) \equiv_{\mathcal{R}_K} stmt(\texttt{if } b \texttt{ then } \{st; \texttt{while } b\ st\} \texttt{ else skip}),$$

which is equivalent to proving that

$$bexp(b) \curvearrowright while(bexp(b), stmt(st)) \equiv_{\mathcal{R}_K} bexp(b) \curvearrowright if(stmt(st) \curvearrowright bexp(b) \curvearrowright while(bexp(b), stmt(st)), nothing).$$

Since $\equiv_{\mathcal{R}_K}$ is defined only using $E$-equivalence classes to prove the latter, it suffices to prove that

$$while(bexp(b), stmt(st)) \equiv_{\mathcal{R}_K} if(stmt(st) \curvearrowright bexp(b) \curvearrowright while(bexp(b), stmt(st)), nothing)$$

We prefer to prove an even more general computational equivalence, namely that for any continuations $k_b$ and $k_{st}$,

$$while(k_b, k_{st}) \equiv_{\mathcal{R}_K} if(k_{st} \curvearrowright k_b \curvearrowright while(k_b, k_{st}), nothing).$$

Let us now use Corollary 1 and show that identifying these two terms has no influence on the data consistency or the computation. Let $e$ be the equation identifying the two terms.

It is relatively easy to see that $e$ does not affect the data consistency. Indeed, one can notice that equations yield structurally equivalent terms, with one exeception, that of the equation defining the value of the computation (yet, that applies only to finished computation, so could not be used in conjunction with $e$ without any rules being applied in between), and thus one can use $e$ to only prove equalities about terms containing the two terms as subterms, i.e., equalities of continuations representing statements, or even programs. In order to destroy the data consistency, one would have to derive equal terms belonging to a syntactic category which also contains visible terms, such as *AExp*, *BExp* or *Pgm*. In fact, not even equations like $\texttt{halt(3)} = \texttt{halt(5)}$ can destroy the data consistency, because there is no way to prove that $E \cup \{\texttt{halt(3)} = \texttt{halt(5)}\} \vdash 3 = 5$.

Let us now show that $E \cup \{e\}$-congruence satisfies the computational granularity condition. Let $t_1$ and $t_2$ be two terms such that $E \cup \{e\} \vdash t_1 = t_2$ and suppose that $\mathcal{R} \vdash t_1 \rightarrow^1 t_1'$. We must find a term $t_2'$ such that $\mathcal{R} \vdash t_2 \rightarrow^1 t_2'$ and $E \cup \{e\} \vdash t_2 = t_2'$.

Let $c_1$ be a context, $\theta_1$ a substitution and $l \rightarrow r$ a rule in $\mathcal{R}$ such that $E \vdash t_1 = c_1[\theta_1(l)]$ and $E \vdash t_1' = c_1[\theta_1(r)]$.

Let us first show that there exists a context $c_2$ and a term $l'$ such that $E \vdash t_2 = c_2[l']$ and $E \cup \{e\} \vdash \theta_1(l) = l'$. Since $E \vdash t_1 = c_1[\theta_1(l)]$ it must be that $E \cup \{e\} \vdash c_1[\theta_1(l)] = t_2$. Notice that, due to the nature of our rewrite rules, $\theta_1(l)$ must be either of form $k(k_{pgm})$ or of the form $k(k_{pgm})$ $store(s)$ for some continuation term $k_{pgm}$ and store term $s$. Since all equations, including $e$, can either apply inside a continuation or are of the form $k(k_1) = k(k_2)$, the deduction for $E \cup \{e\} \vdash c_1[\theta_1(l)] = t_2$ will keep replacing equals by equals either in the $c_1$-part or in the $\theta_1(l)$ part, yielding our affirmation true.

We next show that there exists $r'$ such that $\mathcal{R} \vdash l' \rightarrow r'$ and $E \cup \{e\} \vdash r' = \theta_1(r)$. If $l \rightarrow r$ is not a rule involving *if* or *while*, we can easily replay it for $l'$. Let us exemplify with one of the rules for *and* - others can be treated in a similar manner. Suppose $l \rightarrow r$ is

$$k(true \curvearrowright and(K_2) \curvearrowright K) \rightarrow k(K_2 \curvearrowright K),$$

and suppose $\theta_1(K_2) = k_2$ and $\theta_1(K) = k_{rest}$. Since $e$ makes equal two continuation items which can't be further decomposed without setting them in a proper context, we must have that $E \vdash l' = k(true \curvearrowright and(k'_2) \curvearrowright k'_{rest})$ such that $E \cup \{e\} \vdash k'_2 = k_2$ and $E \cup \{e\} \vdash k'_{rest} = k_{rest}$. Thus, we can also apply $l \to r$ to $c_2[l']$, using the context $c_2$ and the substitution $\theta_2$ given by $\theta_2(K_2) = k'_2$ and $\theta_2(K) = k'_{rest}$, yielding the term $c_2[r'] = c_2[\theta_2(r)] = c_2[k(k'_2 \curvearrowright k'_{rest})]$. Applying the congruence rule we obtain that $E \cup \{e\} \vdash \theta_1(r) = \theta_2(r)$.

If $l \to r$ is one of the *if* rules then we distinguish three cases:
1. if $E \vdash l' = k(t \curvearrowright if(k_1, k_2) \curvearrowright k_{rest})$ for some boolean value $t$, we are in the same situation as above.
2. if $\theta_1(l) = k(true \curvearrowright if(k_1, k_2) \curvearrowright k_{rest})$ and $E \vdash l' = k(true \curvearrowright while(k_b, k_{st}) \curvearrowright k'_{rest})$, then we must also have that $E \cup \{e\} \vdash k_{rest} = k'_{rest}$, $E \cup \{e\} \vdash k_2 = nothing$ and $E \cup \{e\} \vdash k_1 = k_{st} \curvearrowright k_b \curvearrowright while(k_b, k_{st})$. Also, $\theta_1(r) = k(k_2 \curvearrowright k_{rest})$. Using the rule for while for $c_2[l']$, we obtain $t'_2 = c_2[r']$ where $r' = k(k_{st} \curvearrowright k_b \curvearrowright while(k_b, k_{st}) \curvearrowright k'_{rest})$ and using the congruence rule several times we obtain that $\mathcal{E} \cup \{e\} \vdash \theta_1(r) = r'$.
3. if $\theta_1(l) = k(false \curvearrowright if(k_1, k_2) \curvearrowright k_{rest})$ and $E \vdash l' = k(false \curvearrowright while(k_b, k_{st}) \curvearrowright k'_{rest})$, then we must also have that $E \cup \{e\} \vdash k_{rest} = k'_{rest}$ and $E \cup \{e\} \vdash k_2 = nothing$. Also, $\theta_1(r) = k(k_2 \curvearrowright k_{rest})$. Using the rule for while for $c_2[l']$, we obtain $t'_2 = c_2[r']$ where $r' = k(k'_{rest})$ and using the congruence rule several times and the identity equations for *nothing*, we obtain that $\mathcal{E} \cup \{e\} \vdash \theta_1(r) = r'$.

A similar argument can be given if we interchange *if* with *while* above.

We can now show that $E \cup \{e\} \vdash c_1[\theta_1(r)] = c_2[r']$, by replaying those steps of the proof $E \cup \{e\} \vdash c_1[\theta_1(l)] = c_2[l']$ which occur only on positions originating from $c_1$, then using the context rule for equational deduction. □

## 8.2 Relation with Context Reduction

We next show the equivalence between the continuation-based and the context reduction rewriting logic definitions. The specification in Table 9 relates the two semantics, showing that at each computational "point" it is possible to extract from our continuation structure the current expression being evaluated. For each syntactical construct $Syn \in \{AExp, BExp, Stmt, Pgm\}$, we equationally define two (partial) functions:

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

Together, these two functions can be regarded as a parsing process, where the continuation plays the role of "unparsed" syntax, while *Syn* is the abstract syntax tree, i.e., the "parsed" syntax. The formal definitions of *k2Syn* and *kSyn* are given in Table 9.

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

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

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

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

---

$k2Stmt(nothing) = \texttt{skip}$
$k2Stmt(K) = k2Stmt(K'); St \Leftarrow \{K', St\} = kStmt(K) \wedge K' \neq nothing$
$k2Stmt(K) = St \Leftarrow \{K', St\} = kStmt(K) \wedge K' = nothing$
$kStmt(K \curvearrowright write(X)) = \{K', X\texttt{:=}A\} \Leftarrow \{K', A\} = kAExp(K)$
$kStmt(K \curvearrowright while(K_1, K_2)) = \{K', \texttt{if } B \texttt{ then } \{St; \texttt{while } B_1 St\} \texttt{ else skip}\}$
$\quad \Leftarrow \{K', B\} = kBExp(K) \wedge B_1 = k2BExp(K_1) \wedge St = k2Stmt(K_2) \wedge B \neq B_1$
$kStmt(K \curvearrowright while(K_1, K_2)) = \{K', \texttt{while } BSt\}$
$\quad \Leftarrow \{K', B\} = kBExp(K) \wedge B_1 = k2BExp(K_1) \wedge St = k2Stmt(K_2) \wedge B = B_1$
$kStmt(K \curvearrowright if(K_1, K_2)) = \{K', \texttt{if } B \texttt{ then } k2Stmt(K_1) \texttt{ else } k2Stmt(K_2)\}$
$\quad \Leftarrow \{K', B\} = kBExp(K)$
$kStmt(K \curvearrowright halt) = \{K', \texttt{halt } A\} \Leftarrow \{K', A\} = kAExp(K)$

---

$k2AExp(K) = A \Leftarrow \{nothing, A\} = kAExp(K)$
$kAExp(K \curvearrowright kv(Kl, Vl) \curvearrowright K') = kAExp(Vl, K, Kl \curvearrowright K')$

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

---

$k2BExp(K) = B \Leftarrow \{nothing, B\} = kBExp(K)$
$kBExp(K \curvearrowright kv(Kl, Vl) \curvearrowright K') = kBExp(Vl, K, Kl \curvearrowright K')$

$kBExp(K \curvearrowright T) = \{K, T\}$
$kBExp(K \curvearrowright K_1, K_2 \curvearrowright \leq) = \{K, k2AExp(K_1)\texttt{<=}k2AExp(K_2)\}$
$kBExp(K \curvearrowright and(K_2)) = \{K_1, B_1 \texttt{ and } k2BExp(K_2)\} \Leftarrow \{K_1, B_1\} = kBExp(K)$
$kBExp(K \curvearrowright not) = \{K', \texttt{not } B\} \Leftarrow \{K', B\} = kBExp(K)$

Table 9: Recovering the abstract syntax trees from continuations

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 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., 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. But first, the following shows that $\mathcal{R}_{K'}$ does not bring any novelty to $\mathcal{R}_K$, as expected.

**Proposition 5** *For any term $t$ in $\mathcal{R}_K$, $Tree_{\mathcal{R}_K}(t)$ is bisimilar to $Tree_{\mathcal{R}_{K'}}(t)$.*

**Proposition 6** *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 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 $s \simeq \sigma$.*

| n | 15 | 16 | 18 | Memory for 18 |
|---|---|---|---|---|
| ASF+SDF | 1.7 | 2.9 | 11.6 | 13mb |
| BC | 0.3 | 0.6 | 2.8 | <1mb |
| Maude | 3.8 | 7.7 | 31.5 | 6mb |
| Prolog | 2.6 | 5.3 | 23.6 | 600mb |
| Scheme | 3.2 | 6.4 | 25.5 | 5mb |

Table 10: Execution times for Big Step definitions

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* $\sigma'$ *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 $\langle c, \sigma\rangle[x\texttt{:=}i] \to \langle c, \sigma[i/x]\rangle[skip]$ 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: $k(pgm(c[x\texttt{:=}i])\ store(s) \to k(c[\texttt{skip}])\ store(s[x \leftarrow i])$.

2. Actually $\sigma'$ is uniquely determined by $s'$ and $p'$ is the program obtained by advancing $p$ all non-computational steps - which were dissolved by *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 $\mathcal{R}_{K'} \vdash result(k(pgm(c[\texttt{halt}\ i]))\ store(s)) \to i$.

□

# 9 Experiments

We have defined interpretors for the language presented above in two rewrite engines, ASF+SDF (a compiler) and Maude (a fast interpreter with good tool support), in Prolog and in Scheme (modifying existing interpretors used to teach programming languages). Also, the big-step definitions are compared against bc, an C-written interpretor for a subset of C working only with integers. For Prolog we have compiled the programs using the `gprolog` compiler. For Scheme we have used the PLT-Scheme `mred` interpretor. Tests were performed on an Intel Pentium 4@2GHz with 1GB RAM, running Linux. The program chosen to test various implementations consists of $n$ nested loops, each of 2 iterations, parameterized by n. Times are expressed in seconds. A limit of 700mb was set on memory usage, to avoid swapping. For largest runs, peak memory usage was also recorded. For Scheme we have adapted the definitions from [FWH01], chapter 3.9 (evaluation semantics) and 7.3 (continuation based semantics) and a PLT-Redex definition given as example in the installation package (for context reduction).

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

| n | 15 | 16 | 18 | Memory for 18 |
|---|---|---|---|---|
| ASF+SDF | 11.9 | 25.7 | 115.0 | 9mb |
| Maude | 63.4 | 131.2 | 597.4 | 6mb |
| Prolog | 10.2 | 21.9 | - | >700mb |

Table 11: Execution times for Small Step definitions

| n | 9 | 15 | 16 | 18 | Memory for 18 |
|---|---|---|---|---|---|
| ASF+SDF | 0.6 | 88.7 | 214.4 | 1008.6 | 10mb |
| Maude | 3.7 | 552.1 | 1239 | 6142.5 | 6mb |
| Prolog | 0.1 | 10.2 | - | - | >700mb |
| Scheme | 198.2 | - | - | - | >700mb |

Table 12: Execution times for Context Reduction definitions

steps. The style promoted in [FWH01] seems to also take into account efficiency. The only drawback is the fact that it looks more like an interpretor of an SOS definition, the representational distance to the SOS definition being much bigger than in Rewriting Logic/Prolog. The PLT-Redex implementation of context reduction seems to serve more a didactic scope. It compensates lack of speed by nice interface and the possibility to visually trace a run. The rewriting logic implementations seem to be pretty efficient in terms of speed and memory usage (<12mb), while keeping a minimal representational distance to the operational semantics definitions. As expected, continuation based definitions run slower than the big-step, but faster than the others. Nevertheless, we prefer this definitions for the sake of modularity, concurrency and easiness of defining context-intensive features.

# 10 Conclusion and Additional Related Work

We gave a unified overview of how common language operational semantics can be defined as rewrite logic theories. Related big-step and small-step rewrite logic definitions have also been given in [VMO06] and [MB04]. We additionally showed a novel rewrite logic approach to context reduction, together with its equivalence to a more efficient and flexible continuation-based definition. What distinguishes our equivalence result from efforts on efficient implementations of context reduction semantics by CPS transformations to higher-order continuations, e.g., [SF92, DN01, DN05], is that our continuations are *first-order* and our language definitions are *theories in a logic*, with algebraic denotational semantics and complete deduction, not implementations; yet, they can be executed by current rewrite engines at performance comparable with interpreter implementations.

There is much related work on defining programming languages in various computational logical frameworks. We cannot mention all these here, but we refer the interested reader to [Roş05] for a comprehensive discussion. Besides the ones already mentioned, we here list a few others which are, in our view, closer in

| n | 15 | 16 | 18 | Memory for 18 |
|---|---|---|---|---|
| ASF+SDF | 2.5 | 4.7 | 18.3 | 13mb |
| Maude | 8.4 | 15.6 | 63.2 | 7mb |
| Prolog | 4.8 | 9.7 | - | >700mb |
| Scheme | 5.9 | 11.3 | 45.2 | 10mb |

Table 13: Execution times for Continuation based definitions

purpose to our approach: they aim at more than just implementing interpreters. ACL2 [KMM00] allows both (functional) definitions and formal analysis of languages. Abstract State Machine (ASM) [Gur94] 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. There are interesting connections between ASMs and rewriting logic, but their discussion is beyond our scope here. Most of the other approaches ar either (non-trivial) encodings into a (usually typed, higher-order) logic or intepretors (see, for example [FWH01]), which take the formal mathematical definition only as a guideline, usually losing the computational granularity.

*Acknowledgments.* We thank Mark Hills for fruitful discussions and his help in adjusting the PLT-Redex implementation to suit our needs.

# References

[BCD+00] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.4 User Manual*. LORIA, Nancy (France), fourth edition, January 2000.

[BT95] Jan A. Bergstra and J. V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *J. ACM*, 42(6):1194–1230, 1995.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[DF98] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[DN01] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. *Electr. Notes Theor. Comput. Sci.*, 59(4), 2001.

[DN05] Olivier Danvy and Lasse R. Nielsen. Cps transformation of beta-redexes. *Inf. Process. Lett.*, 94(5):217–224, 2005.

[FF86] Matthias Felleisen and Daniel 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, August 1986.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, 2nd edition, 2001.

[GM82] Joseph A. Goguen and José Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notices*, 17(1):9–17, 1982.

[GM92] Joseph A. Goguen and José Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[Gur94] Yuri Gurevich. Evolving algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press, 1994.

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[Kah87] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[KMM00] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[MB04] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2004.

[Mes92] José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[Mil89] R. Milner. *Communication and concurrency*. Prentice Hall, Hemel Hempstead, United Kingdom, 1989.

[MR04] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In David A. Basin and Michaël Rusinowitch, editors, *IJCAR*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.

[MR06] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2006.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

[Roş05] Grigore 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.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *LISP and Functional Programming*, pages 288–298, 1992.

[vdBHKO02] Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[Vis03] Eelco Visser. Program transformation with stratego/xt: Rules, strategies, tools, and systems in stratego/xt 0.9. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003.

[VMO06] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in maude. *J. Log. Algebr. Program.*, 67(1-2):226–293, 2006.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.