A Rewriting Logic Approach to Operational Semantics – Extended Abstract

Traian Florin Şerbănuţă, Grigore Roşu, José Meseguer

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

Abstract

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

1 Introduction

This paper is part of the rewriting logic semantics (RLS) project (see [25,26] and the references there). The broad goal of the project is to develop a tool-supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on *rewriting logic* [22]. Any logical framework worth its salt should be evaluated in terms of its expressiveness and flexibility. Therefore, a very pertinent question is: how does RLS express other approaches to operational semantics? In particular, how well can it express various approaches in the SOS tradition? The goal of this paper is to answer these questions. Partial answers, giving detailed comparisons with specific approaches have appeared elsewhere. For example, [21] and [43] provide comparisons with standard SOS [34]; and [23] compares RLS with both standard SOS and Mosses' modular structural operational semantics (MSOS) [31]. However, no comprehensive comparison encompassing most approaches in the SOS tradition has been given to date. To make our ideas more concrete, in this paper we use a simple programming language, show how it is expressed in each different definitional style, and how that

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

style is captured as a rewrite theory in the RLS framework. We furthermore give correctness theorems showing the faithfulness of the RLS representation for each style. Even though we exemplify the ideas with a simple language for concreteness' sake, the process of representing each definitional style in RLS is completely general and automatable, and in some cases like MSOS has already been automated [6]. The range of styles covered includes: big-step (or natural) SOS semantics; small-step SOS semantics; MSOS semantics; context-sensitive reduction semantics; and continuation-based semantics.

Any logical framework for operational semantics of programming languages has to meet strong challenges. We list below some of them and sketch how they are met in RLS; for a more thorough discussion see [38].

- *Handling of SOS Definitions.* As illustrated in Sections 4, 5, and 6, both big-step and small-step SOS, and also MSOS definitions can be expressed as rewrite theories in RLS; see also [21,43,23].
- *Handling of context-sensitive reduction.* In Section 7 we sketch a general method to express in RLS semantic definitions based on evaluation contexts (e.g., [47]).
- *Handling higher-order syntax.* Higher-order syntax admits first-order representations, e.g., [1,2,40]. Using CINNI [40], all this can be done keeping essentially the original higher-order syntax.
- *Handling continuations.* Continuations [12,35] are traditionally understood as higher-order functions. In Section 8 we present an alternative view of continuations that is intrinsically first-order.
- *Handling concurrency.* One of the strongest points of rewriting logic is precisely that it is a logical framework for concurrency. Unlike standard SOS, which forces an interleaving semantics, true concurrency is directly supported.
- *Expressiveness and flexibility.* RLS does not force on the user any particular definitional style. This is illustrated in this paper by showing how quite different definitional styles can all be faithfully and naturally captured in RLS.
- *Mathematical and operational semantics.* Rewriting logic has both a computational proof theory and an *initial model* semantics, which provides inductive reasoning principles to prove properties. Therefore RLS programming language definitions have *both* an operational rewriting semantics, and a mathematical initial model semantics.
- *Performance Issues.* High-performance systems supporting rewriting can be used to *directly* execute RLS semantic definitions as interpreters. In Section 9 we present encouraging experimental performance results for our example language using various systems and different definitional styles.

Besides the good features mentioned above, another advantage of RLS is the availability of *generic tools* for: (i) syntax; (ii) execution (already mentioned); and (iii) formal analysis. For example, languages such as ASF+SDF [41] and Maude [7] support user-definable syntax. There is a wealth of theorem proving and model checking tools for rewriting/equational-based specifications, which can be used directly to prove properties about language definitions. The fact that these formal analysis tools are generic, should not fool one into thinking that they *must* be in-

efficient. For example, the LTL model checkers obtained for free in Maude from the RLS definitions of Java and the JVM compare favorably in performance with state-of-the-art Java model checkers [11].

Another advantage of RLS is what we call the "abstraction dial," which can be used to reach a good balance between abstraction and computational observability in semantic definitions. The point is which *computational granularity* is appropriate. A small-step semantics opts for very fine-grained computations. But this is not necessarily the only or the best option for all purposes. The fact that an RLS theory's axioms include both equations and rewrite rules provides the useful "abstraction dial," because rewriting takes place *modulo* the equations. That is, computations performed by equations are abstracted out and become *invisible*. This has many advantages, as explained in [25]. For example, in Sections 4 and 5, we use equations to define the semantic infrastructure (stores, etc.) of SOS definitions; in Section 7 equations are also used to hide the extraction and application of evaluation contexts, which are "meta-level" operations, carrying no computational meaning; in Section 8, equations are also used to decompose the evaluation tasks into their corresponding subtasks; finally, in Section 6, equations of associativity and commutativity are used to achievemodularity of language definitions.

2 Rewriting Logic

Rewriting logic [22] is a computational logic that can be efficiently implemented and that has good properties as a general and flexible *logical and semantic framework*, in which a wide range of logics and models of computation can be faithfully represented [21]. In particular, for programming language semantics it provides the RLS framework, of which we here only emphasize the operational semantics aspects.

Two key points are: (i) how rewriting logic combines equational logic and term rewriting; and (ii) what the intuitive meaning of a rewrite theory is. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with Σ a signature, E a set of (conditional) Σ-equations, and R a set of Σ-rewrite rules, with conditions involving both equations and rewrites. That is, a rule in R can have the general form $(\forall X)\ t \longrightarrow t'$ **if** $(\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j)$. Alternatively, such a rule could be displayed with an inference-rule-like notation as $\dfrac{(\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j)}{t \longrightarrow t'}$.

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

Of course, if the equations of an equational theory (Σ, E) are *confluent*, there is another useful representation, namely, as the rewrite theory $(\Sigma, \emptyset, \overrightarrow{E})$, where $\overrightarrow{E}$ are the rewrite rules obtained by orienting the equations E as rules. This representation is at the basis of much work in term rewriting, but by implicitly suggesting that rewrite rules are just an efficient technique for equational reasoning it can blind us to the fact that rewrite rules can have a more general *non-equational* semantics. This is the *raison d'être* of rewriting logic. In rewriting logic a theory $\mathcal{R} = (\Sigma, E, R)$ axiomatizes a *concurrent system*, whose states are elements of the algebraic data type (Σ, E), that is, E-equivalence classes of ground Σ-terms, and whose *atomic transi-*

tions are specified by the rules R. The inference system of rewriting logic allows us to derive as *proofs* all the *concurrent computations* of the system axiomatized by $\mathcal{R}$: concurrent computation and rewriting logic deduction *coincide*.

There are many systems that either specifically implement term rewriting efficiently, so-called as *rewrite engines*, or support term rewriting as part of a more complex functionality. Any of these systems can be used as an underlying platform for execution and analysis of programming languages defined using the techniques proposed in this paper. Without attempting to be exhaustive, we here only mention (alphabetically) some engines that we are more familiar with, noting that many functional languages and theorem provers provide support for term rewriting as well: ASF+SDF [41], CafeOBJ [10], Elan [4], Maude [7], OBJ [16], and Stratego [44]. Some of these engines can achieve remarkable speeds on today's machines, in the order of tens of millions of rewrite steps per second.

3 A Simple Imperative Language

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

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

The result of running a program is the evaluation of AExp in the state after executing Stmt. This BNF syntax is entirely equivalent to an algebraic order-sorted signature having one (mixfix) operation definition per production, terminals giving the name of the operation and non-terminals the arity. For example, the production for if-then-else can be seen as an algebraic 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 use this simple language and will present definitions in various operational semantics styles (big step, small step SOS, MSOS, reduction using evaluation contexts, and continuation-based), as well as the corresponding RLS representation of each definition. We will also characterize the relation between the RLS representations and their corresponding definitional style counterparts, pointing out some strengths and weaknesses for each style. The reader is referred to [19,34,31,47] for details on the described operational semantics styles.

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

Unlike in various operational semantics, which usually abstract stores as func-

tions, 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. 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. We let $s \simeq \sigma$ denote that well-formed state s corresponds to partial function σ.

4 Big-Step Operational Semantics

Introduced as natural semantics in [19], also named relational semantics in [28], 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 the while loop:

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

This rule can be automatically translated into the rewrite rules:

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

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

To give a rewriting logic theory for the big-step semantics, one needs to first define the various configuration constructs, which are assumed by default in *BigStep*, as corresponding operations extending the signature. Then one can define the corresponding rewrite theory $\mathcal{R}_{BigStep}$ entirely automatically.

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

Proposition 4.1 *For any $p \in Pgm$ and $i \in Int$, the following are equivalent: (i) $BigStep \vdash \langle p\rangle \Downarrow \langle i\rangle$; and (ii) $\mathcal{R}_{BigStep} \vdash \langle p\rangle \to^1 \langle i\rangle$.*

The only apparent difference between *BigStep* and $\mathcal{R}_{BigStep}$ is the different notational conventions they use. However, as the above result shows, there is a one-to-one correspondence also between their corresponding "computations" (or executions, or derivations). Therefore, $\mathcal{R}_{BigStep}$ actually *is* the big-step operational semantics *BigStep*, not an "encoding" of it.

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

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

5 Small-Step Operational Semantics

Introduced by Plotkin in [34], also called transition semantics or reduction semantics, small-step semantics captures the notion of one computational step. One inherent technicality involved in capturing small-step operational semantics as a rewrite theory in a one-to-one notational and computational correspondence is that the rewriting relation is by definition transitive, while the small-step relation is *not* transitive (its transitive closure is defined a posteriori). Therefore, we need to devise a mechanism to "inhibit" rewriting logic's transitive and uncontrolled application of rules. An elegant way to achieve this is to view a small step as a modifier of the current configuration. Specifically, we consider "$\cdot$" to be a modifier on the configuration which performs a "small-step" of computation; in other words, we assume an operation $\cdot\_ : \mathit{Config} \to \mathit{Config}$. Then, a small-step semantic rule, e.g. the one for defining while,

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

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

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

As for big-step semantics, the rewriting under context deduction rule for rewriting logic is again inapplicable, since all rules act at the top, on configurations. However, in *SmallStep* it is not the case that all 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 5.1 *For any $p \in \mathit{Pgm}$ and $i \in \mathit{Int}$, $\mathit{SmallStep} \vdash \langle p, \bot\rangle \to^* \langle \texttt{skip}.i, \sigma\rangle$ for some state σ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{eval}(p) \to i$.*

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

Weaknesses. Each small step does the same amount of computation as a big step in finding the next redex. It does not give a "true concurrency" semantics, that is, one has to chose a certain interleaving (no two rules can be applied at the same time), mainly because reduction is forced to occur only at the top. It is still hard to deal with complex 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 to propagate the `halt`. 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*!

6 MSOS Semantics

MSOS [31] was introduced to deal with the non-modularity issues of SOS. The solution proposed in *MSOS* involves moving the non-syntactic state components to the arrow labels, plus a discipline of only selecting needed attributes from states.

A transition in *MSOS* is of the form $P \xrightarrow{u} P'$, where P and P' are program expressions and u is a label describing the structure of the state both before and after the transition. If u is missing, then the state is assumed to stay unchanged. Specifically, u is a record containing fields denoting the semantic components. Modularity is achieved by the record comprehension notation "..." which indicates that more fields could follow but that they are not of interest for this transition. Fields of a label can fall in one of the following categories: *read-only*, *read-write* and *write-only*. *Read-only fields* are only inspected by the rule, but not modified. *Read-write fields* come in pairs, having the same field name, except that the "write" field name is primed. They are used for transitions modifying existing state fields. *Write-only fields* are used to record things not analyzable during the execution of the program, such as the output or the trace. Their names are always primed and they have a free monoid semantics – everything written on then is added at the end. Since the part of the state not involved in a certain rule is hidden through the "..." notation, language extensions can be made modularly. Consider, e.g., adding `halt` to the language. What needs to be done is to add another read-write record field, say *halt?*, along with the possible values *halted(i)*, to signal that the program halted with value i, and *false*, as the default value, along with a construct `stuck` to block the execution of the program.

To represent *MSOS* in rewriting logic, we here follow the methodology in [23]. Using the fact that labels describe changes from their source state to their destination state, one can move the labels back into the configurations. That is, a transition $P \xrightarrow{u} P'$ is modeled as a rewrite step $\cdot\langle P, u^{pre}\rangle \rightarrow \langle P', u^{post}\rangle$, where u^{pre} and u^{post} are *records* describing the state before and after the transition. Note again the use of the "$\cdot$" operator to emulate small steps by restricting transitivity. State records can be specified equationally as wrapping (using a constructor "$\{\_\}$") a set of fields built from *fields* as constructors, using an associative and commutative concatenation operation "$\_,\_$". Fields are built from state attributes; e.g., the store can be embedded into a field by a constructor "$\sigma : \_$". Records u^{pre} and u^{post} are computed from u as follows. For unobservable transitions, $u^{pre} = u^{post}$. *Read-only* fields of u are added to both u^{pre} and u^{post}. *Read-write* fields of u are translated by putting the read part in u^{pre} and the (now unprimed) write part in u^{post}. Notice that the "..." notation gets replaced by a generic field-set variable W. For example, the rules for assignment in MSOS style,

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

are translated into the following rewrite rules (R, R' stand for records and W stands for the remaining of a record):

$$\cdot\langle X\,\texttt{:=}\,A, R\rangle \rightarrow \langle X\,\texttt{:=}\,A', R'\rangle \quad \textbf{if} \quad \cdot\langle A, R\rangle \rightarrow \langle A', R'\rangle$$
$$\cdot\langle X\,\texttt{:=}\,I, \{\sigma : S_0, W\}\rangle \rightarrow \langle \texttt{skip}, \{\sigma : S_0[X \leftarrow I], W\}\rangle$$

Write-only fields $i' = v$ of u are translated as follows: $i : L$, with L a fresh new

variable, is added to u^{pre}, and $i : Lv$ is added to u^{post}. When dealing with observable transitions, both state records meta-variables and ... operations are represented in u^{pre} by some variables, while in u^{post} by others.

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

The relation between *MSOS* and R_{MSOS} definitions assumes that *MSOS* definitions are in a certain *normal form* [23] and is made precise by the following theorem, strongly relating *MSOS* and modular rewriting semantics.

Theorem 6.1 *[23] For each normalized MSOS definition, there is a strong bisimulation between its transition system and the transition system associated to its translation in rewriting logic.*

This translation is the basis for the `Maude-MSOS` tool [6], which was used to define and analyze complex language definitions.

Strengths. As it is a framework on top of any operational semantics, it inherits the strengths of the semantic for which it is used; moreover, it adds to those strengths the important new feature of *modularity*.

Weaknesses. Control is still not explicit in MSOS, making combinations of control-dependent features (e.g., call/cc) harder to specify [31, page 223].

7 Reduction Semantics with Evaluation Contexts

Introduced in [47], also called context reduction, the evaluation contexts style improves over small-step definitional style in two ways: (i) it gives a more compact semantics to context-sensitive reduction, by using parsing to find the next redex rather than small-step rules; and (ii) it provides the possibility of also modifying the context in which a reduction occurs, making it much easier to deal with control-intensive features. For example, defining halt is done now using only one rule, $C[\texttt{halt}\ I] \to I$, preserving the desired computational granularity.

$C ::= [] \mid \langle C, S\rangle$ $\mid$ $\texttt{skip}.C \mid C.A$ $\mid$ $X\texttt{:=}C \mid I + C \mid C + A$ $\dfrac{E \to E'}{C[E] \to C[E']}$	$I_1 + I_2 \to (I_1 +_{Int} I_2)$ $\langle P, \sigma\rangle[X\texttt{:=}I] \to \langle P, \sigma[I/X]\rangle[\texttt{skip}]$ $\texttt{while}\ B\ St \to \texttt{if}\ B\ \texttt{then}\ (St;\texttt{while}\ B\ St)\ \texttt{else skip}$ $C[\texttt{halt}\ I] \to \langle I\rangle$ $C[\texttt{skip}.I] \to \langle I\rangle$

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

$\cdot(I_1 + I_2) \to (I_1 +_{Int} I_2)$
$\cdot(\langle P, S\rangle[X\texttt{:=}I]) \to \langle P, S[X \leftarrow I]\rangle[\texttt{skip}]$
$\cdot(\texttt{while}\ B\ St) \to \texttt{if}\ B\ \texttt{then}\ (St;\texttt{while}\ B\ St)\ \texttt{else skip}$
$\cdot(C[\texttt{halt}\ I]) \to \langle I\rangle[[]]$

$$eval(P) = reduction(\langle P, \emptyset\rangle)$$
$$reduction(Cfg) = reduction(\cdot(Cfg\text{'}))$$
$$reduction(\langle I\rangle) = I$$

Table 1: *CxtRed*-like rules and their corresponding rewriting logic variants

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. 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 "$C[E] \rightarrow C[E']$ when $E \rightarrow E'$," capturing the fact that reductions are allowed to take place only in appropriate evaluation contexts.

Table 1 presents a definition of selected evaluation contexts and some context reduction semantics rules together with their representation within rewriting logic.

By making the evaluation context explicit and changeable, context reduction is, in our view, a significant improvement over small-step SOS. In particular, one can now define control-intensive statements like `halt` *modularly* and at the desired level of computational granularity. Even though the definition gives one the feeling that evaluation contexts and their instantiation come "for free", the application of the "rewrite in context" rule presented above can be expensive in practice. This is because one needs either to parse/search the entire configuration to put it in the form $C[E]$ for some appropriate C satisfying the grammar of evaluation contexts, or to maintain enough information in some special data-structures to perform the split $C[E]$ using only local information and updates. Direct implementations of context reduction such as `PLT-Redex` cannot avoid paying a significant performance penalty, as the performance numbers in Section 9 show [2].

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

The rewrite theory $\mathcal{R}_{CxtRed}$ is obtained by adding the rewrite rules in Table 1 to the equations of *s2c* and *c2s*. The $\mathcal{R}_{CxtRed}$ definition is a faithful representation of context reduction semantics. Also, since parsing issues are abstracted away using equations, the computational granularity is the same, yielding a one-to-one correspondence between the computations performed by the context reduction semantics rules and those performed by the rewriting rules.

Theorem 7.1 *If $s \simeq \sigma$* [3]*, the following hold: (i) $\langle p, \sigma \rangle$ parses in CxtRed as $\langle c, \sigma \rangle[r]$ iff $\mathcal{R}_{CxtRed} \vdash s2c(\langle p, s \rangle) = \langle c, s \rangle[r]$; (ii) $\mathcal{R}_{CxtRed} \vdash c2s(c[r]) = c[r/[]]$ for any valid context c and appropriate redex r; (iii) $CxtRed \vdash \langle p, \sigma \rangle \rightarrow \langle p', \sigma' \rangle$ iff $\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s \rangle) \rightarrow^1 \langle p', s' \rangle$ and $s' \simeq \sigma'$; (iv) $CxtRed \vdash \langle p, \sigma \rangle \rightarrow \langle i \rangle$ iff $\mathcal{R}_{CxtRed} \vdash \cdot(\langle p, s \rangle) \rightarrow^1 \langle i \rangle$; (v) $CxtRed \vdash \langle p, \bot \rangle \rightarrow^* \langle i \rangle$ iff $\mathcal{R}_{CxtRed} \vdash eval(p) \rightarrow i$.*

Strengths. Context reduction semantics distinguishes small-step rules into computational rules and rules needed to find the redex (the latter are transformed into

[2] Refocusing [9] proposed automatically generating abstract machines for overcoming this problem.

[3] We let $s \simeq \sigma$ denote the fact that equationally defined state s represents the store σ.

grammar rules generating the allowable contexts). This makes definitions more compact. It improves over small step semantics by allowing the context to be changed by execution rules. It can deal easily with control-intensive features.

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

8 A Continuation-Based Semantics

The idea of continuation-based interpreters for programming languages and their relation to abstract machines has been well studied (see, for example, [12]). In this section we propose a rewriting logic semantics based on a structure that provides a *first-order* representation of continuations; this is the only reason why we call this structure a "continuation"; but notice that it can just as well be regarded as a post-order representation of the abstract syntax tree of the program, so one needs no prior knowledge of continuations [12] in order to understand this section. We will show the equivalence of this theory to the context reduction semantics theory.

Based on the desired order of evaluation, the program is sequentialized by transforming it into a list of tasks to be performed in order. This is done once and for all at the beginning, the benefit being that at any subsequent moment in time we know precisely where the next redex is: at the top of the tasks list.

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. The continuation is defined as a list of tasks, where the list constructor "_ ↷ _" is associative, having as identity a constant "*nothing*". We also use lists of values and continuations, each having an associative list append constructor "_,_" with identity ".". We use variables K and V to denote continuations and values, respectively; also, we use Kl and Vl for lists of continuations and values, respectively. We call the list of tasks a *continuation* because it resembles the idea of continuations as higher-order functions. However, our continuation is a pure first order flattening of the program. For example $aexp(A_1 + A_2) = (aexp(A_1), aexp(A_2)) \curvearrowright +$ precisely encodes the order of evaluation: first A_1, then A_2, then sum the values. Also, $stmt(\texttt{while}\ B\ St) = bexp(B) \curvearrowright while(bexp(B), stmt(St))$ says that the loop is dependent on the value of B for its evaluation. *pgm*, *stmt*, *bexp*, *aexp* are used to flatten the program to a continuation, taking into account the order of evaluation [4]. 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 gives the possibility of achieving "true concurrency", since rules which do not act on the same parts of the context can be applied in parallel. We here only discuss the sequential variant of our continuation-based semantics, because our language is sequential. In

[4] The effect of these functions is somehow similar to what one would obtain in a higher order world by means of CPS transformations [46] or conversions to monadic normal form [30].

[36] we show how the same technique can be used, with no additional effort, to define concurrent languages: as expected, one continuation structure is generated for each concurrent thread or process. Then rewrite rules can apply "truly concurrently" at the tops of continuations. Table 2 presents some rewrite rules from the continuation-based definition of our language.

$aexp(A_1 + A_2) = (aexp(A_1), aexp(A_2)) \curvearrowright +$	$k((I_1, I_2) \curvearrowright + \curvearrowright K) \to k(I_1 +_{Int} I_2 \curvearrowright K)$
$stmt(X := A) = aexp(A) \curvearrowright write(X)$	
	$k(I \curvearrowright write(X) \curvearrowright K)\ store(Store) \to k(K)\ store(Store[X \leftarrow I])$
$stmt(\texttt{while}\ B\ St) = bexp(B) \curvearrowright while(bexp(B), stmt(St))$	$k(false \curvearrowright while(K_1, K_2) \curvearrowright K) \to k(K)$
	$k(true \curvearrowright while(K_1, K_2) \curvearrowright K) \to k(K_2 \curvearrowright K_1 \curvearrowright while(K_1, K_2) \curvearrowright K)$
$stmt(\texttt{halt}\ A) = aexp(A) \curvearrowright halt$	$k(I \curvearrowright halt \curvearrowright K) \to k(I)$
$pgm(St.A) = stmt(St) \curvearrowright aexp(A)$	
$\langle P \rangle = result(k(pgm(P))\ store(empty))$	$result(k(I)\ store(Store)) = I$
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 into two rules:	
$k((Vl, Ke, Kel) \curvearrowright K) = k(Ke \curvearrowright (Vl, nothing, Kel) \curvearrowright K)$	
$k(V \curvearrowright (Vl, nothing, Kel) \curvearrowright K) = k((Vl, V, Kel) \curvearrowright K)$	

Table 2: Rewriting logic theory $\mathcal{R}_K$ (continuation-based definition of the language)

There exists a close connection between definitions of languages using reduction semantics with evaluation contexts and the style promoted in this section:

Theorem 8.1 *Suppose $s \simeq \sigma$. Then: (i) 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 $\to^{\leq 1} = \to^0 \cup \to^1$; (ii) If $\mathcal{R}_K \vdash k(pgm(p))\ store(s) \to k(k')\ store(s')$ then there exists p' and σ' such that $CxtRed \vdash \langle p, \sigma \rangle \to^* \langle p', \sigma' \rangle$, $\mathcal{R}_K \vdash k(pgm(p')) = k(k')$ and $s' \simeq \sigma'$; (iii) $CxtRed \vdash \langle p, \bot \rangle \to^* i$ iff $\mathcal{R}_K \vdash \langle p \rangle \to i$ for any $p \in Pgm$ and $i \in Int$.*

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

Weaknesses. The program is now flattened in the continuation; several new operations (continuation constants) need to be introduced, which "replace" the corresponding original language constructs.

9 Experiments

RLS specifications can be turned into interpreters for the specified language. To analyze the efficiency of this approach, we wrote the RLS definitions above in two rewrite engines, namely `ASF+SDF 1.5` (a compiler) and `Maude 2.2` (a fast interpreter with good tool support), and in several programming languages with built-in support for matching, namely Haskell, Ocaml and Prolog. For each definitional style tested (except small-step SOS), we have included for comparison interpreters

in Scheme, adapting definitions from [13], 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). Since RLS representation of MSOS relies intensively on matching modulo associativity and commutativity, which is only supported by Maude, we performed no experiments for it.

One tested program consists of 15 nested loops, each of 2 iterations. The other program verifies Collatz' conjecture up to 300. The following table gives for each definitional style the running time of the various interpreters (first column – nested loops; second column – Collatz). Times are expressed in seconds. A limit of 700mb was set on memory usage; "-" found in a table cell means the memory limit was reached; an empty cell means the combination was not attempted. For Haskell we used the `ghc` compiler. For Ocaml we used the `ocamlcopt` compiler. For Prolog we compiled the programs using `gprolog`. For Scheme we used the PLT-Scheme interpreter. Tests were done on a Pentium 4@2GHz with 1GB RAM, running Linux.

Language	Bigstep		SmallStep		Reduction		Continuations	
ASF+SDF	1.7	265.1	11.9	769.6	88.7	891.3	2.5	344.7
Haskell	0.3	32.1	3.2	167.4	5.8	157.2	0.6	41.1
Maude	3.8	184.5	63.4	>1000	552.1	>1000	8.4	483.9
Ocaml	0.5	10.2	1.0	21.0	1.8	11.0	0.5	10.9
Prolog	1.6	-	7.0	-	9.4	-	3.0	-
Scheme	3.8	122.3			-	-	5.9	323.6

Table 3: Experiments (times in seconds)

Prolog yields pretty fast interpreters. However, for backtracking reasons, it needs to maintain the stack of all predicates tried on the current path, thus the amount of memory grows with the number of computational steps. The style promoted in [13] seems to also take into account efficiency. The only drawback is the fact that it looks more like an interpreter of a big-step definition, the representational distance to the big-step definition being much bigger than in interpreters based on RLS. The PLT-Redex implementation of context reduction ran out of memory for the presented inputs (for 9 nested loops it finished in 198 seconds). The rewriting logic implementations seem to be quite efficient in terms of speed and memory usage, while keeping a minimal representational distance to the operational semantics definitions. In particular, RLS definitions interpreted in Maude are comparable in terms of efficiency with the interpreters in Scheme, while having the advantage of being formal definitions.

10 Related Work

There is much related work on frameworks for defining programming languages. Without trying to be exhaustive, we mention some of them.

Algebraic denotational semantics. This approach, (see [17,42] for two recent books), is a special case of RLS, namely, the case in which the rewrite theory $\mathcal{R}_{\mathcal{L}}$ defining language $\mathcal{L}$ is an equational theory. While algebraic semantics shares a number of advantages with RLS, its main limitation is that it is not well-suited for concurrent language definitions.

Other RLS work. RLS is a collective international project. Through the efforts of various researchers, there is by now a substantial body of work demonstrating the

usefulness of this approach. A first snapshot of the RLS project was given in [26], and a second in [25]. This paper can be viewed as third snapshot focusing on the variety of definitional styles supported. A substantial body of experience in giving programming language definitions, and using those definitions both for execution and for analysis purposes has already been gathered; an up-to-date list of references on RLS can be found in the companion tech report [38].

Higher-order approaches. The most classic higher-order approach, although not exactly operational, is *denotational semantics* [37]. Denotational semantics has some similarities with its first-order algebraic cousin mentioned above, since both are based on semantic equations. Two differences are: (i) the use of first-order equations in the algebraic case versus the higher-order ones in traditional denotational semantics; and (ii) the kinds of models used in each case. A related class of higher-order approaches uses higher-order functional languages or higher-order theorem provers to give operational semantics to programming languages. Without trying to be complete, we can mention, for example, the use of Scheme in [13], the use of ML in [33], and the use of Common LISP within the ACL2 prover in [20]. There is also a body of work on using monads [29,45] to implement language interpreters in higher-order functional languages; the monadic approach has better modularity characteristics than standard SOS. A third class of higher-order approaches are based on the use of higher-order abstract syntax (HOAS) and higher-order logical frameworks, such as LF or λ-Prolog [32], to encode languages as logical systems. For recent work in this direction see [27] and references there.

Other approaches. Going back to the Centaur project [5,8], logic programming has been used as a framework for SOS definitions. Note that λ-Prolog [32] belongs both in this category and in the higher-order one. Abstract State Machine (ASM) [18] can encode any computation and have a rigorous semantics, so any programming language can be defined as an ASM and thus implicitly be given a semantics. Both big- and small-step ASM semantics have been investigated. The semantics of various programming languages, including, for example, Java [39], has been given using ASMs. The Chemical Abstract Machine [3] avoids some of the limitations of SOS in defining concurrent programming languages and was introduced in the same journal volume as Rewriting Logic; as shown in [22], any chemical abstract machine definition is a rewrite logic theory. Tile logic [14] also supports definitions of concurrent languages and has been compared to and translated into rewriting logic [24,15].

11 Conclusions

We have tried to show how RLS can be used as a logical framework for operational semantics definitions of programming languages. By showing how it can faithfully capture big-step and small-step SOS, MSOS, context reduction, and continuation-based semantics, we hope to have illustrated what might be called its *ecumenical* character: flexible support for a wide range of definitional styles, without forcing or pre-imposing any given style. We think that this flexibility makes RLS useful as a way of exploring *new* definitional styles. For highly-concurrent languages, such as mobile languages, or for languages involving concurrency, real-time and/or probabilities, a centralized approach forcing an interleaving semantics is unnatural.

We have, of course, refrained from putting forward any specific suggestions in this regard. But we think that new definitional styles are worth investigating; and hope that RLS in general, and this paper in particular, will stimulate such investigations.

References

[1] Abadi, M., L. Cardelli, P.-L. Curien and J.-J. Lévy, *Explicit Substitutions*, in: *Proc. POPL'90* (1990), pp. 31–46.

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

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

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

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

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

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

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

[9] Danvy, O. and L. R. Nielsen, *Refocusing in reduction semantics*, Technical Report BRICS RS-04-26, University of Aarhus (2004).

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

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

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

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

[14] Gadducci, F. and U. Montanari, *The tile model*, in: G. Plotkin, C. Stirling and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT, 2000 Also, TR-96-27, C.S. Dept., Univ. of Pisa, 1996.

[15] Gadducci, F. and U. Montanari, *Comparing logics for rewriting: Rewriting logic, action calculi and tile logic*, Theoret. Comput. Sci. **285** (2002), pp. 319–358.

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

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

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

[19] Kahn, G., *Natural semantics.*, in: F.-J. Brandenburg, G. Vidal-Naquet and M. Wirsing, editors, *STACS*, Lecture Notes in Computer Science **247** (1987), pp. 22–39.

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

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

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

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

[24] Meseguer, J. and U. Montanari, *Mapping tile logic into rewriting logic*, in: F. Parisi-Presicce, editor, *Proceedings of WADT'97, 12th Workshop on Recent Trends in Algebraic Development Techniques*, LNCS **1376** (1998), pp. 62–91.

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

[26] Meseguer, J. and G. Rosu, *Rewriting logic semantics: From language specifications to formal analysis tools.*, in: D. A. Basin and M. Rusinowitch, editors, *IJCAR*, Lecture Notes in Computer Science **3097** (2004), pp. 1–44.

[27] Miller, D., *Representing and reasoning with operational semantics.*, in: U. Furbach and N. Shankar, editors, *IJCAR*, Lecture Notes in Computer Science **4130** (2006), pp. 4–20.

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

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

[30] Moggi, E., *Notions of computation and monads*, Inf. Comput. **93** (1991), pp. 55–92.

[31] Mosses, P. D., *Modular structural operational semantics*, J. Log. Algebr. Program. **60–61** (2004), pp. 195–228.

[32] Nadathur, G. and D. Miller, *An overview of λProlog*, in: K. Bowen and R. Kowalski, editors, *Fifth Int. Joint Conf. and Symp. on Logic Programming* (1988), pp. 810–827.

[33] Pierce, B., "Types and Programming Languages," MIT Press, 2002.

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

[35] Reynolds, J. C., *The Discoveries of Continuations*, LISP and Symbolic Computation **6** (1993), pp. 233–247.

[36] Roşu, G., *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).

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

[38] Şerbănuţă, T. F., G. Roşu and J. Meseguer, *A rewriting logic approach to operational semantics*, Technical Report UIUCDCS-R-2007-2820, University of Illinois, Department of COmputer Science (2007).

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

[40] Stehr, M.-O., *CINNI - a generic calculus of explicit substitutions and its application to lambda-, sigma- and pi-calculi* (2000), proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.

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

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

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

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

[45] Wadler, P., *The essence of functional programming.*, in: *POPL*, 1992, pp. 1–14.

[46] Wand, M., *Continuation-based program transformation strategies.*, J. ACM **27** (1980), pp. 164–180.

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