Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 373 (2007) 213–237
Theoretical Computer Science
www.elsevier.com/locate/tcs

The rewriting logic semantics project

José Meseguer, Grigore Roşu*

Department of Computer Science, University of Illinois at Urbana-Champaign, United States

Abstract

Rewriting logic is a flexible and expressive logical framework that unifies algebraic denotational semantics and structural operational semantics (SOS) in a novel way, avoiding their respective limitations and allowing succinct semantic definitions. The fact that a rewrite logic theory's axioms include both equations and rewrite rules provides a useful "abstraction dial" to find the right balance between abstraction and computational observability in semantic definitions. Such semantic definitions are directly executable as interpreters in a rewriting logic language such as Maude, whose generic formal tools can be used to endow those interpreters with powerful program analysis capabilities.
© 2007 Published by Elsevier B.V.

Keywords: Semantics and analysis of programming languages; Rewriting logic

1. Introduction

The fact that rewriting logic [40,9] provides an easy and expressive way to develop executable formal definitions of languages, which can then be subjected to different tool-supported formal analyses, is by now well established [6, 70,67,65,42,68,15,58,69,27,25,37,7,44,45,12,11,26,21,59,1,66,23,60,38,36,30].

In fact, the just-mentioned papers by different authors are contributions to a collective ongoing research project which we call the *rewriting logic semantics project*. A first snapshot of this project was given in [45]. In our view, what makes this project promising is the combination of three interlocking facts:

(1) that, as explained in Sections 1.1 and 1.2, and further substantiated in the rest of this paper, rewriting logic is a flexible and expressive *logical framework* that unifies algebraic denotational semantics and SOS in a novel way, avoiding their respective limitations and allowing very succinct semantic definitions;
(2) that rewriting logic semantic definitions are *directly executable* in a rewriting logic language such as Maude [17], and can thus become efficient interpreters; and

* Corresponding author. Tel.: +1 217 244 7431; fax: +1 217 244 6869.
E-mail address: grosu@cs.uiuc.edu (G. Roşu).

0304-3975/$ - see front matter © 2007 Published by Elsevier B.V.
doi:10.1016/j.tcs.2006.12.018

(3) that *generic formal tools* such as the Maude LTL model checker [24], the Maude inductive theorem prover [19,20], and new tools such as a language-generic partial order reduction tool [26], allow us to amortize tool development cost across many programming languages, that can thus be endowed with powerful program analysis capabilities; furthermore, *genericity does not imply inefficiency*: in some cases the analyses so obtained outperform those of well-known language-specific tools [27,25].

1.1. Semantics: Algebraic vs. SOS

Two well-known semantic frameworks for programming languages are: algebraic denotational semantics and structural operational semantics (SOS).

In *algebraic denotational semantics* (see [72,32,8,49] for early papers and [31] for a recent book), equational specifications are used as *semantic equations* to give a formal semantics to a language. This use of semantic equations is similar to the one in traditional denotational semantics [62,63,61,50]. 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. Both variants are *denotational*, so that syntactic entities are mapped to their semantic interpretations in a compositional way. In the algebraic case, initial algebra semantics pioneered by Joseph Goguen is the preferred approach (see, for example, [32,31]), but other approaches, based on loose semantics or on final algebras, are also possible.

Strong points of algebraic denotational semantics include: (1) it is a *model-theoretic* semantics, consisting of models of interest in the initial algebra semantics approaches, or of all models in the *loose* approaches; (2) it also has a *proof-theoretic*, operational semantics, given by *equational reduction* with the semantic equations; (3) semantic definitions can be turned into interpreters, thanks to efficient first-order equational languages (ACL2, OBJ, ASF + SDF, Maude, etc.); (4) there is good first-order theorem proving support.

However, algebraic denotational semantics shares the following drawbacks with traditional denotational semantics: (1) it is well suited for *deterministic* languages such as conventional sequential languages or purely functional languages, but it is quite poorly suited to define the semantics of *concurrent languages*: one can *indirectly model*[1] some concurrency aspects with devices such as a scheduler, or lazy data structures, but a direct comprehensive modeling of all concurrency aspects remains elusive; and (2) semantic equations are typically *unmodular*, i.e., adding new features to a language often requires *extensive redefinition* of earlier semantic equations.

In SOS, formal definitions take the form of *semantic rules*. SOS is a proof-theoretic approach, focusing on giving a detailed step-by-step (which can be "small" or "big") formal description of a program's execution. The semantic rules are used as inference rules to reason about what computation steps are possible. Typically, the rules follow the syntactic structure of programs, defining the semantics of a language construct in terms of that of its parts. The *locus classicus* is Plotkin's Aarhus lectures [55]; there is again a vast literature on the topic that we do not attempt to survey; for a good textbook introduction see [35].

Strong points of SOS include: (1) it is a general, yet quite intuitive formalism, allowing detailed modeling of program executions; (2) it has a simple *proof-theoretic* semantics using semantic rules as inference rules; (3) it is fairly well suited to model *concurrent languages* (only if using a "small-step" methodology), and can also deal well with the detailed execution of deterministic languages; (4) it allows *mathematical reasoning and proof*, by reasoning inductively or coinductively about the inference steps.

However, SOS has the following drawbacks: (1) in its standard formulation it imposes a centralized *interleaving semantics* of concurrent computations, which may be unnatural in some cases (for example for highly decentralized and asynchronous mobile computations); this problem is avoided in "reduction semantics", which is different from SOS and is in fact a special case of rewriting semantics (see Section 2.1); (2) standard SOS definitions are notoriously *unmodular*, unless one adopts Mosses' MSOS framework [51–53]; (3) although some tools have been built to execute SOS definitions (see for example [22,34,54]), tool support for verifying properties is perhaps less developed than for denotational semantics.

[1] Two good examples of indirectly modeling concurrency within a purely functional framework are the ACL2 semantics of the JVM using a scheduler [48], and the use of lazy data structures in Haskell to analyze cryptographic protocols [2].

1.2. Abstraction dial: Unifying algebraic denotational semantics and SOS

For the most part, algebraic denotational semantics and SOS have lived separate lives. Pragmatic considerations and differences in taste tend to dictate which framework is adopted in each particular case. For concurrent languages, SOS is superior and tends to prevail as the formalism of choice; for deterministic languages, algebraic approaches are widely used. Of course, there are also practical considerations of tool support for execution and formal reasoning.

In the end, algebraic denotational semantics and SOS, although each very valuable in its own way, are "single hammer" approaches. Would it be possible to seamlessly *unify* them within a more flexible and general framework? Could their respective limitations be overcome when they are thus unified? Our proposal is that rewriting logic [40, 9] does indeed provide one such unifying framework. The key to this unification is what we call rewriting logic's *abstraction dial*. The point is that in algebraic denotational semantics, entities are *identified by the semantic equations* and have unique *abstract denotations* in the corresponding models. In our dial metaphor, this means that in algebraic denotational semantics the abstraction dial is *always turned all the way up to its maximum position*. By contrast, one of the key features of SOS is providing a very detailed, step-by-step formal description of a language's evaluation mechanisms. As a consequence, most entities – except perhaps for built-in data, stores and environments, which are typically treated on the side – are *primarily syntactic*, and computations are described in full detail. In our metaphor, this means that in SOS the abstraction dial is *always turned down to its minimum position*.

How is the unification and corresponding availability of an abstraction dial achieved? Roughly speaking,[2] a rewrite theory is a triple (Σ, E, R), with (Σ, E) an equational theory with Σ a signature of operations and sorts, and E a set of (possibly conditional) equations, and with R a set of (possibly conditional) rewrite rules. Equational semantics is obtained as the special case in which $R = \varnothing$, so we only have the semantic equations E and the abstraction dial is turned up to its maximum position. Roughly speaking, SOS (with unlabeled transitions) is obtained as the special case in which $E = \varnothing$, and we only have (possibly conditional) rules R rewriting purely syntactic entities (terms), so that the abstraction dial is turned down to the minimum position.

Rewriting logic's "abstraction dial" is precisely its distinction between equations E and rules R in a rewrite theory (Σ, E, R). *States of the computation* are then E-equivalence classes, i.e., *abstract elements* in the initial algebra $T_{\Sigma/E}$. Because of rewriting logic's "Equality" inference rule (Section 2) a rewrite with a rule in R is understood as a transition $[t] \longrightarrow [t']$ between such abstract states. The dial, however, can be turned up or down. We can turn it *all the way down to its minimum* by converting all equations into rules, transforming (Σ, E, R) into $(\Sigma, \varnothing, R \cup E)$. This gives us the most concrete, SOS-like semantic description possible.[3] What can we do in general to make a specification *as abstract as possible*, that is, to "turn the dial up" as much as possible? We can identify a subset $R_0 \subseteq R$ such that: (1) $R_0 \cup E$ is Church-Rosser on the terms of interest (well-formed programs); and (2) R_0 is the biggest possible with this property. In actual language specification practice this is not hard to do. We illustrate this idea with a simple example language in Section 3.1. Essentially, we can use semantic equations for most of the sequential features of a programming language: only when language features could lead to nondeterminism (particularly if the language has threads and/or processes) or for intrinsically concurrent features are rules (as opposed to equations) really needed. The use of rules versus equations in a rewriting logic language semantics definition will be further discussed in Section 3.1.

The conceptual distinction between equations and rules also has important practical consequences for *program analysis*, because it affords a massive *state space reduction* which can make formal analyses such as breadth-first search and model checking enormously more efficient. Because of state-space explosion, such analyses could easily become unfeasible if we were to use an SOS-like specification in which all computation steps are described with rules. This capacity of dealing with abstract states is a crucial reason why our generic tools, when instantiated to a given programming language definition, tend to result in program analysis tools of competitive performance.

[2] We postpone discussion of "equational reduction strategies" μ and "frozen" argument information ϕ to Section 2. In more detail, a rewrite theory will be axiomatized as a tuple $(\Sigma, E, \mu, R, \phi)$.

[3] Thanks to the influence of reduction semantics (see Section 2.1), the idea of adding some syntactic congruences (such as associativity and commutativity) to SOS rules has been gradually adopted. However, such congruences are purely *structural*, while in rewriting logic semantic definitions many equations carry *computational* meaning.

Of course, the price to pay in exchange for abstraction is a *coarser level of granularity* in respect to what aspects of a computation are *observable* at that abstraction level. For example, when analyzing a sequential program using a semantics in which most sequential features have been specified with equations, all sequential subcomputations will be abstracted away, and the analysis will focus on memory and thread interactions. If a finer analysis is needed, we can often obtain it by "turning down the abstraction dial" to the right observability level by *converting some equations into rules*. That is, we can regulate the dial to find for each kind of analysis the best possible balance between abstraction and observability. Note that very fine and subtle distinctions are possible in a language definition in our framework. The equations in E may contain: (i) *structural* equations, such as associativity, commutativity, identity, or explicit substitution equations for binding operators; (ii) *infra-structural* equations, aimed at defining operations and data-types useful for maintaining the state infrastructure of the programming language; and (iii) *computational* equations corresponding to deterministic computation steps.

This paper is organized as follows. Background on membership equational logic and rewriting logic is given in Section 2. The relationship to algebraic denotational semantics and SOS is discussed in greater detail in Section 2.1. We then illustrate our ideas by giving a rewriting logic semantics to a simple programming language in Section 3.1, summarize other language specification case studies in Section 3.2, and present a specialized notation that can also be used for compiler generation in Section 3.3. Program analysis techniques and tools are discussed in Section 4, including search and model checking analyses (4.1), as well as abstract-semantics-based analyses (4.2) and deductive approaches (4.3). We end with some concluding remarks in Section 5.

2. Rewriting logic semantics

Membership equational logic. A membership equational logic (MEL) [41] *signature* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A MEL Σ-algebra A contains a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1\cdots k_n,k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, while elements without a sort are *errors*. We write $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ to denote, respectively, the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables. Given a MEL signature Σ, *atomic formulae* have either the form $t = t'$ (Σ-equation) or $t : s$ (Σ-membership) with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$; and Σ-*sentences* are conditional formulae of the form "$(\forall X)\ \varphi$ if $\bigwedge_i p_i = q_i \ \wedge\ \bigwedge_j w_j : s_j$", where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i, and w_j are in X. A MEL theory is a pair (Σ, E) with Σ a MEL signature and E a set of Σ-sentences. We refer to [41] for the detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, and initial and free algebras. In particular, given a MEL theory (Σ, E), its initial algebra is denoted $T_{\Sigma/E}$; its elements are E-equivalence classes of ground terms in T_Σ. Order-sorted notation $s_1 < s_2$ can be used to abbreviate the conditional membership "$(\forall x : k)\ x : s_2$ if $x : s_1$". Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom "$(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s$ if $\bigwedge_{1\le i\le n} x_i : s_i$". We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of "$(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t'$ if $\bigwedge_{1\le i\le n} x_i : s_i$".

For execution purposes we typically impose some requirements on a MEL theory. First of all, its sentences may be decomposed as a union $E \cup A$, with A a set of equations that we will reason *modulo* (for example, A may include associativity, commutativity and/or identity axioms for some of the operators in Σ). Second, the sentences E are typically required to be Church-Rosser[4] modulo A, so that we can use the conditional equations E as equational rewrite rules modulo A. Third, for some applications it is useful to make the equational rewriting relation[5] *context-sensitive*. This can be accomplished by specifying a function $\mu : \Sigma \longrightarrow \mathbb{N}^*$ assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a list $\mu(f) = i_1 \ldots i_k$ of *argument positions* , with $1 \le i_j \le n$, which must be fully evaluated (up to the context-sensitive equational reduction strategy specified by μ) in the order specified by

[4] See [5] for a detailed study of equational rewriting concepts and proof techniques for MEL theories.

[5] As we shall see, in a rewrite theory $\mathcal{R}$ rewriting can happen at two levels: (1) *equational rewriting* with (possibly conditional) equations E; and (2) *non-equational rewriting* with (possibly conditional) rewrite rules R. These two kinds of rewriting *are different*. Therefore, to avoid confusion we will always qualify rewriting with equations as *equational rewriting*.

the list $i_1 \dots i_k$ before applying any equations whose left-hand sides have f as their top symbol. For example, for $f = \mathit{if\_then\_else\_fi}$ we may give $\mu(f) = \{1\}$, meaning that the first argument must be fully evaluated before the equations for $\mathit{if\_then\_else\_fi}$ are applied.[6] Therefore, for execution purposes we can specify a MEL theory as a triple $(\Sigma, E \cup A, \mu)$, with A the axioms we rewrite modulo, and with μ the map specifying the context-sensitive equational reduction strategy.

Rewrite theories. A *rewriting logic specification or theory* is a tuple $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$, with: (1) $(\Sigma, E \cup A, \mu)$ a MEL theory with "modulo" axioms A and context-sensitive equational reduction strategy μ; (2) R a set of *labeled conditional rewrite rules* of the general form

$$r : (\forall X)\ t \longrightarrow t' \ \text{ if } \ \left(\bigwedge_i u_i = u_i'\right) \wedge \left(\bigwedge_j v_j : s_j\right) \wedge \left(\bigwedge_l w_l \longrightarrow w_l'\right) \tag{1}$$

where the variables appearing in all terms are among those in X, terms in each rewrite or equation have the same kind, and in each membership $v_j : s_j$ the term v_j has kind $[s_j]$; and (3) $\phi : \Sigma \longrightarrow \mathcal{P}(\mathbb{N})$ a mapping assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a set $\phi(f) = \{i_1, \dots, i_k\}$, $1 \le i_1 < \cdots < i_k \le n$ of *frozen argument positions*[7] under which it is forbidden to perform any rewrites.

Intuitively, $\mathcal{R}$ specifies a *concurrent system*, whose states are elements of the initial algebra $T_{\Sigma/E\cup A}$ specified by $(\Sigma, E \cup A)$, and whose *concurrent transitions* are specified by the rules R, subject to the frozenness requirements imposed by ϕ. The frozenness information is important in practice to forbid certain rewritings. For example, when defining the rewriting semantics of a process calculus, one may wish to require that in prefix expressions $\alpha.P$ the operator $\_.\_$ is *frozen in the second argument*, that is, $\phi(\_.\_) = \{2\}$, so that P cannot be rewritten under a prefix. The frozenness idea can be extended to variables in terms as follows: given a Σ-term $t \in T_\Sigma(X)$, we call a variable $x \in \mathit{vars}(t)$ *frozen* in t iff there is a nonvariable position $\alpha \in \mathbb{N}^*$ such that $t/\alpha = f(u_1, \dots, u_i, \dots, u_n)$, with $i \in \phi(f)$, and $x \in \mathit{vars}(u_i)$. Otherwise, we call $x \in X$ *unfrozen*. Similarly, given Σ-terms $t, t' \in T_\Sigma(X)$, we call a variable $x \in X$ *unfrozen* in t and t' iff it is unfrozen in both t and t'.

Note that a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, \phi, R)$ specifies two kinds of *context-sensitive* rewriting requirements: (1) equational rewriting with E modulo A is made context-sensitive by μ; and (2) non-equational rewriting with R is made context-sensitive by ϕ. But the maps μ and ϕ impose *different types* of context-sensitive requirements: (1) $\mu(f)$ specifies a list of arguments that *must be* fully evaluated with the equations E (up to the strategy μ) before equations for f are applied; and (2) $\phi(f)$ specifies arguments that *must never be* rewritten with the rules R under the operator f. The maps μ and ϕ substantially increase the expressive power of rewriting logic for semantic definition purposes, because various order-of-evaluation and context-sensitive information, which would have to be specified by explicit rules in a formalism like SOS, becomes implicit and is encapsulated in μ and ϕ.

Rewriting logic deduction. Given $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$, 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 $f : k_1 \dots k_n \longrightarrow k$ in Σ, with $\{1, \dots, n\} - \phi(f) = \{j_1, \dots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \le i \le n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \le l \le m$,

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

[6] Maude has a functional sublanguage whose modules are membership equational theories. Maps μ specifying context-sensitive equational reduction strategies are called *evaluation strategies* [17], and $\mu(f) = i_1 \dots i_k$ is specified with the `strat` keyword followed by the string $(i_1\ \dots\ i_k\ 0)$, with 0 indicating evaluation at the top of the function symbol f.

[7] In Maude, $\phi(f) = \{i_1, \dots, i_k\}$ is specified by declaring f with the `frozen` attribute, followed by the string $(i_1\ \dots\ i_k)$.

- **Replacement**. For each $\theta : X \longrightarrow T_\Sigma(Y)$ with, say, $X = \{x_1, \ldots, x_n\}$, and $\theta(x_l) = p_l$, $1 \leq l \leq n$, and for each rule in R of the form,

$$q : (\forall X)\ t \longrightarrow t' \ \text{ if } \ \left(\bigwedge_i u_i = u_i'\right) \wedge \left(\bigwedge_j v_j : s_j\right) \wedge \left(\bigwedge_k w_k \longrightarrow w_k'\right)$$

with $Z = \{x_{j_1}, \ldots, x_{j_m}\}$ the set of unfrozen variables in t and t', then,

$$\left(\bigwedge_r (\forall Y)\ p_{j_r} \longrightarrow p'_{j_r}\right)$$

$$\frac{\left(\bigwedge_i (\forall Y)\ \theta(u_i) = \theta(u_i')\right) \wedge \left(\bigwedge_j (\forall Y)\ \theta(v_j) : s_j\right) \wedge \left(\bigwedge_k (\forall Y)\ \theta(w_k) \longrightarrow \theta(w_k')\right)}{(\forall Y)\ \theta(t) \longrightarrow \theta'(t')}$$

where for $x \in X - Z$, $\theta'(x) = \theta(x)$, and for $x_{j_r} \in Z$, $\theta'(x_{j_r}) = p'_{j_r}$, $1 \leq r \leq m$.

- **Transitivity**. $\frac{(\forall X)\ t_1 \longrightarrow t_2 \quad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$.

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 left-hand side to the corresponding right-hand side instance, the state fragments in the substitution of the rule's variables can also be rewritten, provided the variables involved are not frozen. Finally, the "Transitivity" rule allows us to build longer concurrent computations by composing them sequentially.

For execution purposes a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ should satisfy some basic requirements. These requirements are assumed to hold by a rewriting logic language such as Maude. First, in the MEL theory $(\Sigma, E \cup A, \mu)$ E should be ground Church-Rosser modulo A – for A a set of equational axioms for which matching modulo A is decidable – and ground terminating modulo A up to the context-sensitive strategy μ. Second, the rules R should be *coherent* with E modulo A [71]; intuitively, this means that, to get the effect of rewriting in equivalence classes modulo $E \cup A$, we can always first simplify a term with the equations E to its canonical form modulo A, and then rewrite with a rule in R. Finally, the rules in R should be *admissible* [17], meaning that in a rule of the form (1), besides the variables appearing in t there can be extra variables in t', provided that they also appear in the condition and that they can all be *incrementally instantiated* by either matching a pattern in a "matching equation" or performing breadth first search in a rewrite condition (see [17] for a detailed description of admissible equations and rules).

A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ has both a *deduction-based operational semantics*, and an *initial model denotational semantics*. Both semantics are defined naturally out of the proof theory described in Section 2. The deduction-based operational semantics of $\mathcal{R}$ is defined as the collection of *proof terms* [40,9] of the form $\alpha : t \longrightarrow t'$. A proof term α is an algebraic description of a proof tree proving $\mathcal{R} \vdash t \longrightarrow t'$ by means of the inference rules of Section 2. A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ has also a model theory. The models of $\mathcal{R}$ are *categories* with a $(\Sigma, E \cup A)$-algebra structure [40,9]. The class of models of a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ has an *initial model* $\mathcal{T}_\mathcal{R}$ [40,9]. The initial model semantics is obtained as a *quotient* of the just-mentioned deduction-based operational semantics, precisely by axiomatizing algebraically when two proof terms $\alpha : t \longrightarrow t'$ and $\beta : u \longrightarrow u'$ denote the same concurrent computation.

2.1. Rewriting logic semantics of programming languages

Rewriting logic's operational and denotational semantics apply in particular to the specification of programming languages. We define the semantics of a (possibly concurrent) programming language, say $\mathcal{L}$, by specifying a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, (E \cup A)_{\mathcal{L}}, \mu_{\mathcal{L}}, R_{\mathcal{L}}, \phi_{\mathcal{L}})$, where $\Sigma_{\mathcal{L}}$ specifies $\mathcal{L}$'s *syntax* and the auxiliary operators (store, environment, etc.), $(E \cup A)_{\mathcal{L}}$ specifies the semantics of all the *deterministic features* of $\mathcal{L}$ and of the auxiliary semantic operations, the rewrite rules $R_{\mathcal{L}}$ specify the semantics of all the *concurrent features* of $\mathcal{L}$, and $\mu_{\mathcal{L}}$ and $\phi_{\mathcal{L}}$ specify additional context-sensitive rewriting requirements for the equations $(E \cup A)_{\mathcal{L}}$ and the rules $R_{\mathcal{L}}$. Section 3.1 gives a detailed case study of a rewriting semantics $\mathcal{R}_{\mathcal{L}}$ for $\mathcal{L}$ a simple programming language.

The relationships with equational semantics and SOS can now be described more precisely. First of all, note that when $R = \varnothing$, the only possible arrows are identities, so that the initial model $\mathcal{T}_{\mathcal{R}}$ becomes isomorphic to the initial algebra $T_{\Sigma/E \cup A}$. That is, traditional initial algebra semantics [33], which is the most commonly used form of algebraic denotational semantics, appears as a special case of rewriting logic's initial model semantics.

As already mentioned, we can also obtain SOS as the special case in which we "turn the abstraction dial" all the way down to the minimum position by turning all equations into rules. Intuitively, an SOS rule of the form

$$\frac{P_1 \longrightarrow P_1' \quad \ldots \quad P_n \longrightarrow P_n'}{Q \longrightarrow Q'}$$

corresponds to a rewrite rule with *rewrites in its condition*

$$Q \longrightarrow Q' \ \text{ if } \ P_1 \longrightarrow P_1' \ \wedge \ \cdots \ \wedge \ P_n \longrightarrow P_n'.$$

There are however some technical differences between the meaning of a transition $P \longrightarrow Q$ in SOS and a sequent $P \longrightarrow Q$ in rewriting logic. In SOS a transition $P \longrightarrow Q$ is always a *one-step*[8] transition. Instead, because of "Reflexivity" and "Transitivity", a rewriting logic sequent $P \longrightarrow Q$ may involve many rewrite steps; furthermore, because of "Congruence", such steps may correspond to rewriting subterms. These technical differences present no real difficulty for faithfully expressing SOS within rewriting logic: as shown in detail in [45], we can just "dumb down" the rewriting logic inference to force one-step rewrites in conditions. This can be easily accomplished by adding two auxiliary operators $[\_]$ and $\langle \_ \rangle$, so that SOS rules of the form above can be exactly simulated by conditional rewrite rules of the form

$$[Q] \longrightarrow \langle Q' \rangle \ \text{ if } \ [P_1] \longrightarrow \langle P_1' \rangle \ \wedge \ \cdots \ \wedge \ [P_n] \longrightarrow \langle P_n' \rangle.$$

For example, a "big-step" SOS definition of a conventional programming language may contain rules of the form

$$\frac{E_1 \longrightarrow V_1, \ E_2 \longrightarrow V_2}{E_1 + E_2 \longrightarrow V_1 + V_2},$$

where values (V_1, V_2, $V_1 + V_2$, etc.) are particular expressions. Inhibiting the transitivity in rewriting logic as above, the rewrite rule corresponding to this SOS rule is:

$$[E_1 + E_2] \longrightarrow \langle V_1 + V_2 \rangle \ \text{ if } \ [E_1] \longrightarrow \langle V_1 \rangle \ \wedge \ [E_2] \longrightarrow \langle V_2 \rangle.$$

To state that values are special cases of expressions (which is *not* always the case in language definitions—e.g., closures as values corresponding to function declarations are not expressions), one also needs to add an unconditional rule $[V] \longrightarrow \langle V \rangle$.

In general, SOS rules may have *labels*, *decorations*, and *side conditions*. In fact, there are many SOS rule variants and formats. For example, additional semantic information about stores or environments can be used to decorate an SOS rule. Therefore, showing in detail how SOS rules in each particular variant or format can be faithfully represented by corresponding rewrite rules would be a tedious business. Fortunately, Peter Mosses, in his modular structural operational semantics (MSOS) [51–53], has managed to neatly pack all the various pieces of semantic information usually *scattered throughout* a standard SOS rule *inside labels on transitions*, where now labels have a record structure

[8] The step can be "small" or "big", depending on the style of the semantics. But there is no transitivity by default in SOS definitions. If transitivity is needed, it has to be defined either as an additional SOS rule, or as a meta-notation for a transitive closure relation $\longrightarrow^{\star}$.

whose fields correspond to the different semantic components (the store, the environment, action traces for processes, and so on) *before and after* the transition thus labeled is taken. The paper [44] defines a faithful representation of an MSOS specification $\mathcal{S}$ as a corresponding rewrite theory $\tau(\mathcal{S})$, provided that the MSOS rules in $\mathcal{S}$ are in a suitable normal form.

A different approach, also subsumed by rewriting logic semantics, is sometimes described as *reduction semantics*. It goes back to Berry and Boudol's Chemical Abstract Machine (Cham) [4], and has been used to give semantics to different concurrent calculi and programming languages (see [4,47] for two early references). In essence, a reduction semantics, either of the Cham type or with a different choice of basic primitives, can be naturally seen as a special type of rewrite theory $\mathcal{R} = (\Sigma, A, R, \phi)$, where A consists of *structural axioms*, e.g., associativity and commutativity of multiset union for the Cham, and R is a set of *unconditional* rewrite rules. The frozenness information ϕ is specified by giving explicit inference rules, stating which kind of *congruence* is permitted for each operator for rewriting purposes. *Evaluation context semantics* [28] is a variant of reduction semantics in which the applicability of reductions is controlled by requiring them to occur in definable *evaluation contexts*. In rewriting logic one can obtain a similar effect by making use of the frozenness information. However, the rewriting logic specification style is slightly different, because operations are assumed congruent by default: one only needs to explicitly state which operations are *not* congruent (or frozen) and for which arguments.

3. Specifying programming languages

There can be many different styles to *specify* the same system or design in rewriting logic, depending upon one's goals, such as operational efficiency, verification of properties, mathematical clarity, modularity, or just one's personal taste. It is therefore not surprising that different, semantically equivalent rewriting logic definitional styles are possible for specifying a given programming language $\mathcal{L}$. However, what is common to all these styles is the fact that there is a sort `State`, together with appropriate constructors to store state information needed to define the various language constructs, such as locations, values, environments, stores, etc., as well as means to define the two important semantic aspects of each language construct, namely: (1) the value it evaluates to in a given state; and (2) the state resulting after its evaluation.

3.1. A simple example

In this section we illustrate a *continuation-based* definitional style by means of SIMPLE, a Simple IMPerative LanguagE. SIMPLE is a C-like language, whose programs consist of function declarations. The execution of SIMPLE programs starts by calling the function `main()`. Besides allowing (recursive) functions and other common language features (loops, assignments, conditionals, local and global variables, etc.), SIMPLE is a *multithreaded* programming language, allowing its users to dynamically create, destroy and synchronize threads. We only focus on the important definitional aspects here. The interested reader can consult [46] for a complete definition of SIMPLE, as well as more details on our language definitional methodology. Our other continuation-based definitions of programming languages are very similar to the definition of SIMPLE below, though some languages contain significantly more features and, for some features, several cases need to be considered.

The specification of each language feature consists of two subparts, its *syntax* and its *semantics*. We define each of the two subparts as separate Maude modules, the latter importing the former. For clarity, we prefer to first define all the syntactic components of the language features, then the necessary state infrastructure, and finally the semantic components.

SIMPLE *syntax*. Since Maude provides a parser generator for user-defined, context-free[9] mix-fix syntax, we can define the syntax of our programming languages in Maude and use its parser generator to parse programs. We

[9] A context-free grammar can be specified as an order-sorted signature Σ: the sorts exactly correspond to nonterminals; and the mix-fix operator declarations and subsort declarations exactly correspond to grammar productions. Since in Maude each module is either a MEL theory or a rewrite theory, its signature part Σ specifies a user-defined context-free grammar for which Maude automatically generates a parser. To keep parsing efficient, Maude's MSCP parser has certain characteristics and limitations (e.g., parsing at token rather than at character level) that we do not discuss here, but refer the interested reader to [17].

show below how to define the syntax of SIMPLE using mix-fix notation. In Maude, functional modules (i.e., ones with initial semantics) are defined using the keywords `fmod ... is ... endfm`; modules are imported (without semantic constraints) using the keyword `including`. We start by defining *names*, or *identifiers*, which will be used as variable or function names. Maude's built-in `QID` module provides us with an unbounded number of quoted identifiers, e.g., `'abc123`, so we can import those and declare `Qid` a subsort of `Name`. Besides the quoted identifiers, one can also define several names as constants, so one can omit the quotes:

```
fmod NAME is including QID .
  sort Name .  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Name .
endfm
```

In the definition of Java [27,25], for example, to give the user meaningful feedback when parsing of large programs fails, we used an external parser for Java programs. If one uses an external parser, then one can easily either generate for each identifier in the program a constant of sort `Name` as above, or quote all the identifiers in the program.

SIMPLE is an *expression language*, meaning that everything evaluating to a value parses to an expression[10]; in other words, we do *not* decide on particular types of particular language constructs (bool, int, function, etc.) at parse time. As discussed in Section 4.2, complex type checkers can be easily defined on top of the expression syntax if needed. By making use of sorts, it would be straightforward to define different syntactic categories, such as statements, arithmetic and boolean expressions, etc. We first define expressions generically as terms of sort `Exp` extending names and Maude's built-in integers. At this moment we do not need/want to know what other language constructs will be added later on:

```
fmod GENERIC-EXP-SYNTAX is including NAME .  including INT .
  sort Exp .  subsorts Int Name < Exp .
endfm
```

We are now ready to add language features to the syntax of SIMPLE. We start by adding common arithmetic expressions:

```
fmod ARITHMETIC-EXP-SYNTAX is including GENERIC-EXP-SYNTAX .
  ops _+_ _-_ _*_ : Exp Exp -> Exp [ditto] .
  ops _/_ _%_ : Exp Exp -> Exp [prec 31] .
endfm
```

To save space, from here on we omit adding the entire module defining a particular feature, but only mention its important characteristics; see [46] for a complete definition of SIMPLE. Let us next add syntax for boolean expressions:

```
  ops true false : -> Exp .
  ops _==’_ _!=’_ _<’_ _>’_ _<=’_ _>=’_ : Exp Exp -> Exp [prec 37] .
  op _and_ : Exp Exp -> Exp [prec 55] .
  op _or_ : Exp Exp -> Exp [prec 59] .
  op not_ : Exp -> Exp [prec 53] .
```

Note that we do not distinguish between arithmetic and boolean expressions at this stage. This will be considered when we define the semantics of SIMPLE. The attribute `ditto` associated to some of the arithmetic operators says that they inherit the attributes of the previously defined operators with the same name; these operators were imported together with the built-in `INT` module. Built-in modules/features are, of course, not necessary in a language definition. However, it is very convenient to reuse existing efficient libraries for basic language features, such as integer arithmetic, instead of defining them from scratch.

Overloading built-in operators is practically useful, but it can sometimes raise syntactic/parsing problems. E.g., the built-in binary relational operators on integers evaluate to sort `Bool`, which, for technical reasons, we do *not* want to define as a subsort of `Exp`. The technical reasons are due to incidental overloading of some operations on both integers and booleans (such as exclusive or); note, however, that Maude's booleans and integers were not designed to be collapsed under one common supersort. Consequently, we cannot overload those operators in our SIMPLE language; indeed, otherwise an expression like "`3 < 5`" could have both sorts `Bool` and `Exp`, so Maude would rightfully report

[10] There is exception for the sort of programs, `Pgm`, defined below; programs also evaluate to values, but for clarity we prefer to use a sort different from `Exp`.

ambiguous parsing error messages. That is the reason why we added a back-quote to their names above. If one uses an external parser, like we did in [27,25] in the context of Java, then one can easily quote these operations at parse time.

Conditionals are indispensable in almost any programming language:

```
op if_then_ : Exp Exp -> Exp .
op if_then_else_ : Exp Exp Exp -> Exp .
```

Assignments and sequential composition are core features of an imperative language. Unlike in C, we prefer to use the less confusing `:=` operator for assignments (as opposed to just `=`, which we could have used without a need to quote it, but which many consider to be a poor notation for assignment):

```
  op _:=_ : Name Exp -> Exp [prec 41] .
...
  op _;_ : Exp Exp -> Exp [assoc prec 100] .
```

The attribute[11] `assoc` says that the operation is associative. This is an essentially semantic property; however, we prefer to give it as part of the syntax because Maude's parser makes use of it to eliminate the need for parentheses.

Lists are used several times in the definition of SIMPLE: lists of names are needed for variable and function declarations, lists of expressions are needed for function calls, lists of values are needed for output as a result of the execution of a program. There are at least two generic approaches to define lists algebraically. One uses *parameterized modules* and another uses a *"subtype polymorphism" methodology*. The former requires special support for parameterization, while the latter requires the specification language to support subsorts. Maude supports both parameterization and subsorting, so either approach can be chosen. In our context of language definitions, lists of elements of sorts related by subsorting are needed, such as, e.g., lists of names and lists of expressions; since one wants lists of elements of the subsort to be a subsort of lists of elements of the supersort, even if one chooses a parametric approach to lists, one still needs to define various subsorting relations among the various lists of elements of related sorts. Moreover, subtle (pre-)regularity aspects of an order-sorted signature in the context of associativity [18] require us to define "intersection" sorts and lists on them for any two sorts having a common supersort. These suggest that the second approach to defining lists is more appropriate in our context. We first define the basic module for lists:

```
fmod LIST is sort List .
  op nil : -> List .
  op _,_ : List List -> List [assoc id: nil prec 99] .
endfm
```

The initial model of the specification above is not interesting, because it only contains one element, `nil`. However, this module will be extended shortly; the initial models of its extensions will contain actual lists of elements of desired sorts. Each time we need lists of a particular sort `S`, all we need to do is to define a sort `SList` extending the sort `List` above, together with an overloaded comma operator. In particular, we can define lists of names as follows:

```
fmod NAME-LIST is including NAME .  including LIST .
  sort NameList .  subsorts Name List < NameList .
  op '(') : -> NameList .  eq () = nil .
  op _,_ : NameList NameList -> NameList [ditto] .
endfm
```

As syntactic sugar, note that we defined an additional empty list of names operator, `()`, with the same semantics as `nil`. This is because we prefer to write `f()` instead of `f(nil)` when defining or calling functions without arguments. Blocks allow one to group several statements into just one statement. Additionally, blocks can define local variables for temporary use:

```
  op {} : -> Exp .
  op {_} : Exp -> Exp .
  op {local_;_} : NameList Exp -> Exp [prec 100] .
```

[11] In Maude the "modulo axioms" A in a MEL theory $(\Sigma, E \cup A, \mu)$ or a rewrite theory $\mathcal{R}$ can include any combination of *associativity*, *commutativity*, and *identity* axioms. They are declared as equational attributes of their corresponding operator with the `assoc`, `comm`, and `id:` keywords. The Maude interpreter then supports rewriting modulo such axioms with equations and rules.

The above general definition of blocks does not only provide the user with a powerful construct allowing on-the-fly variable declarations; but it will also ease later on the definition of functions: a function's body is just an ordinary expression; if one needs local variables then one just defines the body of the function to be a block with local variables. The syntax of loops and print is straightforward. We allow both `for` and `while` loops:

```
op for(_;_;_)_ : Exp Exp Exp Exp -> Exp .
op while__ : Exp Exp -> Exp .
op print_ : Exp -> Exp .
```

Lists of expressions will be needed shortly to define function calls:

```
fmod EXP-LIST is including GENERIC-EXP-SYNTAX .  including NAME-LIST .
  sort ExpList .  subsort Exp NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm
```

We are now ready to define the syntax of functions. Each function has a name, a list of parameters, and a body expression. A function call is a name followed by a list of expressions. Functions can be enforced to return abruptly with a typical `return` statement. As explained previously, programs should provide a function called `main`, which is where the execution starts from:

```
sort Function .
op function___ : Name NameList Exp -> Function [prec 115] .
op __ : Name ExpList -> Exp [prec 0] .
op return : Exp -> Exp .
op main : -> Name .
```

A program can have more functions, which can even be mutually recursive. We define syntax for *sets* of functions. We use sets because their order does not matter at all: each function can see all the other declared functions in its environment:

```
sort FunctionSet .  subsort Function < FunctionSet .
op empty : -> FunctionSet .
op __ : FunctionSet FunctionSet -> FunctionSet
          [assoc comm id: empty] .
```

We want to allow dynamic thread creation in SIMPLE, together with some appropriate synchronization mechanism. The `spawn` statement takes any expression and starts a new thread evaluating that expression. Following common sense in multithreading, the child thread inherits the environment of its parent thread; thus, data-races start becoming possible. To avoid race conditions and to allow synchronization in our language, we introduce a simple lock-based policy, in which threads can acquire and release locks:

```
ops (lock_) (spawn_) (acquire_) (release_) : Exp -> Exp .
```

We have defined the syntax of all the desired language features of SIMPLE. All that is needed now to define the syntax of programs is to put all these definitions together. A program consists of a set of global variable declarations and of a set of function declarations:

```
fmod SIMPLE-SYNTAX is
  including ARITHMETIC-EXP-SYNTAX .
  ...
  sort Pgm .  subsort FunctionSet < Pgm .
  op global_;_ : NameList FunctionSet -> Pgm [prec 122] .
endfm
```

To test the syntax one can parse programs that one would like to execute/analyze later on, when the semantics will also be defined. In our experience, this is a good moment to write tens of benchmark programs. For example, the following concurrent program is a SIMPLE version of a deadlock-prone dining philosophers' program. It parses as a well-formed program (when requested to parse, as below, using the Maude command `parse`). Once we define the semantics of SIMPLE, we will be able not only to execute this multithreaded program but also to analyze it, thus detecting a deadlock automatically:

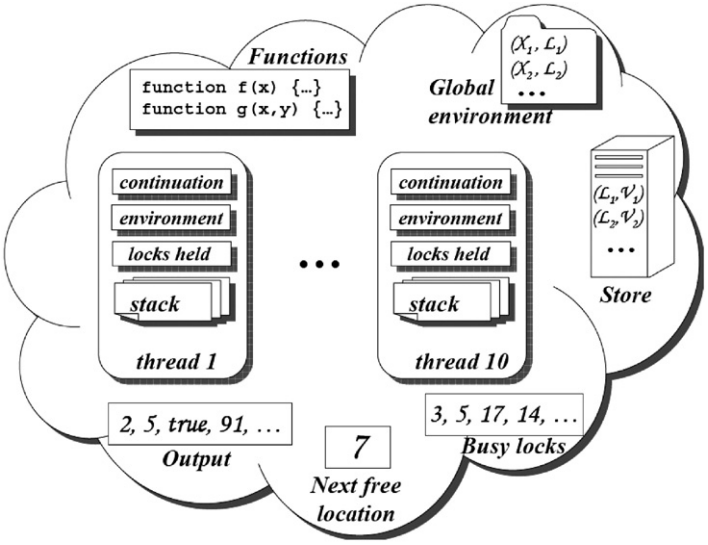

Fig. 1. SIMPLE state infrastructure.

```
parse (
  global n ;

  function f(x) {
    acquire lock(x) ;
      acquire lock(x + 1) ;
        --- eat
      release lock(x + 1) ;
    release lock(x)
  }

--- go to right column
```

```
function main() {
  local i ;
  n := 3 ;
  for(i := 1 ; i <’ n ; i := i + 1)
    spawn(f(i)) ;
  acquire lock(n) ;
    acquire lock(1) ;
      --- eat
    release lock(1) ;
  release lock(n)
}
) .
```

SIMPLE ***state infrastructure.*** In many language definitional methodologies, including the one in this paper, computation is regarded as a sequence of transitions between *states*. We here think of states rather abstractly, in the sense that a state can be any term. For example, in a definition of a pure functional language a state can be any so-called (in SOS terminology) “configuration” (containing mappings of names into values, a particular expression to evaluate, etc.), while in a definition of a message-passing language a state can be the (multi-)set of all processes with their local states, as well as all the pending messages, the existing resources, etc. The semantics of the various language constructs is defined in terms of how they use or change an existing state. Consequently, before we can proceed to define the semantics of SIMPLE we need to first define its entire state infrastructure. In our approach, the state can be regarded as a “nested soup”, its ingredients being formally called *state attributes*. By “soup” we here mean a multiset with associative and commutative union, and by “nested” we mean that certain attributes can themselves contain other soups inside (for example the threads). Fig. 1 shows graphically the state structure of SIMPLE that we consider here. We next informally describe each of the state ingredients, without giving formal Maude definitions (the interested reader is referred to [46] for details):

Store. The store is a mapping of *locations* into *values*. Formally, a binary operation “`[_,_] : Location Value -> Store`” is defined, together with an associative and commutative operation “`__ : Store Store -> Store`”, as well as appropriate equations guaranteeing that no two distinct pairs have the same location. Operations

“`_[_] : Store Location -> Value`” and
“`_[_<-_] : Store Location Value -> Store`”

for look-up and update, respectively, are also defined as part of the store’s interface. Each thread will contain its own environment mapping names into locations. Two or more threads can all have access in their environments to the same location in the store, thus potentially causing data-races.

Global environment. The global environment maps each global name into a corresponding location. Locations for the global names will be allocated once and for all at the beginning of the execution.

Functions. To facilitate (mutually) recursive function definitions, each function sees all the other functions defined in the program. An easy way to achieve this is to simply keep the set of functions as part of the state.

Next free location. This is a natural number giving the next location available to assign a value to in the store. This is needed in order to know where to allocate space for local variables in blocks. Note that in this paper we do not consider garbage collection (otherwise, a more complex schema for the next free location would be needed).

Output. The values printed with the `print` statement are collected in an output list. This list will be the result of the evaluation of the program.

Busy locks. Thread synchronization in SIMPLE is based on locks. Locks can be acquired or released by threads. However, if a lock is already taken by a thread, then any other thread acquiring the same lock is blocked until the lock is released by the first thread. Consequently, we need to maintain a list of locks that are already busy (taken by some threads); a thread can acquire a lock only if that lock is not in the list of busy locks.

Threads. Each thread needs to maintain its own state, because each thread may execute its own code at any given moment and can have its own resources (locations it can access, locks held, etc.). The state of each thread will contain the following ingredients:

- *Continuation.* The tasks/code to be executed by each thread will be encoded as a continuation structure. A *continuation* is generally understood as a means to encode the remaining part of the computation. We use the operation "`_->_ : ContinuationItem Continuation -> Continuation`" to place a new item on top of an existing continuation. If `K` is some continuation and `V` is some value, then the term `val(V) -> K` is read as "the value `V` is passed to the continuation `K`, which hereby knows how to continue the computation". Several continuation items, such as "`val(V)`" will be defined modularly as we give the semantics of the various language features.
- *Environment.* A thread may allocate local variables during its execution. The thread can use these variables in addition to the global ones. The local environment of a thread assigns to each variable that the thread has access to a unique location in the store.
- *Locks held.* A set of locks held by each thread needs to be maintained. When a thread is terminated, all locks it holds must be released.
- *Stack.* The execution of a thread may naturally involve (recursive) function calls. To ease the definition of the `return` statement, it is convenient to "freeze" and stack the current control context whenever a function is called. Then `return` simply pops the previous control context.

Once all the state ingredients above are defined formally (see [46]), one can formalize the entire state infrastructure as a "nested soup" of such ingredients:

```
fmod SIMPLE-STATE is
  sorts SimpleStateAttribute SimpleState SimpleThreadStateAttribute SimpleThreadState .
  subsort SimpleStateAttribute < SimpleState .
  subsort SimpleThreadStateAttribute < SimpleThreadState .
... including environment, store, etc ...
  op empty : -> SimpleState .
  op __ : SimpleState SimpleState -> SimpleState [assoc comm id: empty] .
  op empty : -> SimpleThreadState .
  op __ : SimpleThreadState SimpleThreadState -> SimpleThreadState [assoc comm id: empty] .
  op t : SimpleThreadState -> SimpleStateAttribute .
  op k : Continuation -> SimpleThreadStateAttribute .
  op stack : Continuation -> SimpleThreadStateAttribute .
  op holds : CounterSet -> SimpleThreadStateAttribute .
  op nextLoc : Nat -> SimpleStateAttribute .
  op mem : Store -> SimpleStateAttribute .
  op output : IntList -> SimpleStateAttribute .
  op globalEnv : Env -> SimpleStateAttribute .
  op busy : IntSet -> SimpleStateAttribute .
  op functions : FunctionSet -> SimpleStateAttribute .
endfm
```

Many sorts and operations above are constructors for (nested multi-)sets. The operation `t` is a constructor for the top state "soup"; it takes a thread state and "wraps" it into a state attribute. Similarly, the operation `k` wraps a continuation into a state attribute, `busy` a set of integers (busy locks), and so on. It is easy to note that the signature in the module above is nothing but an algebraic encoding of the state infrastructure in Fig. 1.

SIMPLE *semantics.* We can now start defining the semantics of SIMPLE. Some operations will be used frequently in the definition of a language, so we define them once and for all at the beginning. The continuation items `exp` and `val` below will be used in the semantics of almost all the language constructs:

```
op exp : ExpList Env -> ContinuationItem .
op val : ValueList -> ContinuationItem .
```

The meaning of `exp(E, Env)` on top of a continuation `K`, that is, the meaning of `exp(E, Env) -> K`, is that `E` is the very next "task" to evaluate, in the environment `Env`. Once the expression `E` evaluates to some value `V`, the continuation item `val(V)` is placed on top of the continuation `K`, which will further process it. It is actually going to be quite useful to extend the meaning above to lists of (sequentially-evaluated) expressions and values, respectively:

```
var El : ExpList .  var Vl : ValueList .
eq k(exp(nil, Env) -> K) = k(val(nil) -> K) .
eq k(exp((E,E',El), Env) -> K) = k(exp(E, Env) -> exp((E',El), Env) -> K) .
eq k(val(V) -> exp(El, Env) -> K) = k(exp(El, Env) -> val(V) -> K) .
eq k(val(Vl) -> val(V) -> K) = k(val(V,Vl) -> K) .
```

Note that the expressions `El` in `exp(El, Env)` on top of a continuation are evaluated *sequentially*. Since `El` will typically contain the subexpressions of a language construct, that means that we impose *apriori* an order of evaluation of these subexpressions. This is a reasonable convention in many languages and an artifact of many continuation-based language definitional techniques. If one does *not* want to impose an order of evaluation and wants to allow full non-determinism in the evaluation of these expressions, then one needs to slightly change the structure of the state to allow *nested continuations*: the task of evaluating `El` would split into a "soup" of tasks, one for each expression in `El`; once each expression is evaluated, their values are combined back into a result list. In other words, the evaluation of each expression would be regarded as a new "thread"; indeed, such a semantic definition would resemble quite closely the one for threads that is discussed in the sequel.

There are typically several statements in a programming language that write values to particular locations in the store. Note that the operation of writing a value at a location needs to be a *rewrite rule*, as opposed to an equation. This is because different threads or processes may "compete" to write the same location at the same time, with different choices potentially making a drastic difference in the overall behavior of the program:

```
op writeTo_ : Location -> ContinuationItem .
rl t(k(val(V) -> writeTo(L) -> K) TS) mem(Mem) => t(k(K) TS) mem(Mem[L <- V]) .
```

Therefore, if a value `V` is passed on top of a continuation to `writeTo(L)`, then `V` is written in the store at location `L` and no value is placed as a result on the top of the continuation.

Like writing values in the store, binding values to names is also a crucial operation in a language definition. Defining this operation involves several steps, such as creating new locations, binding the new names to them in the current environment, and finally writing the values to the newly created locations. It is interesting to note that, despite the fact that binding involves writing the store, it can be completely accomplished using just equations. What makes this possible is the fact that the behavior of a program does/should *not* depend upon which particular location is allocated to a new name:

```
op bindTo : NameList Env -> ContinuationItem .
op env : Env -> ContinuationItem .
var TS : SimpleThreadState .
eq t(k(val(V,Vl) -> bindTo((X,Xl),Env) -> K) TS) mem(Mem) nextLoc(N)
 = t(k(val(Vl) -> bindTo(Xl,Env[X <- loc(N)]) -> K)TS) mem(Mem[loc(N),V]) nextLoc(N+1) .
eq k(val(nil) -> bindTo(Xl, Env) -> K) = k(bindTo(Xl, Env) -> K) .
eq t(k(bindTo((X,Xl), Env) -> K) TS) nextLoc(N)
 = t(k(bindTo(Xl, Env[X <- loc(N)]) -> K) TS) nextLoc(N + 1) .
eq k(bindTo(nil, Env) -> K) = k(env(Env) -> K) .
op exp* : Exp -> ContinuationItem .
eq env(Env) -> exp*(E) -> K = exp(E, Env) -> K .
```

The above `env` operator allows one to temporarily "freeze" a certain environment in the continuation; `exp*` applied to an expression `E` grabs the environment `Env` frozen in the continuation and generates the task `exp(E,Env)`.

The remaining modules define the continuation-based semantics of the various SIMPLE language constructs, in the same order in which we introduced their syntax previously. As before, we only mention the important parts of each module, ignoring tedious module and variable declarations. We next define the semantics of generic expressions, i.e., integers and names. An integer expression evaluates to its integer value, while a name needs to first grab its location from the environment and then its value from the store. Note that the evaluation of a variable, in other words its "read" action, needs to be a rewrite rule rather than an equation. This is because for SIMPLE programs a read of a variable may compete with writes of the same variable by other threads, with different orderings leading to potentially different behaviors:

```
op int : Int -> Value .
eq k(exp(I, Env) -> K) = k(val(int(I)) -> K) .
rl t(k(exp(X, Env) -> K) TS) mem(Mem) => t(k(val(Mem[Env[X]]) -> K) TS) mem(Mem) .
```

The continuation-based semantics of arithmetic expressions is straightforward. For example, in the case of the expression `E + E'` on top of the current continuation, one generates the task `(E,E')` on the continuation, followed by the task "add them" (formally a continuation item constant `+`). Once the list `(E,E')` is processed (using other equations or rules), i.e., evaluated to a list of values, in our case of the form `(int(I), int(I'))`, then all that is left to do is to combine these values into a result value for the original expression, in our case `int(I + I')`, and place it on top of the continuation[12]:

```
op + : -> ContinuationItem .
eq k(exp(E + E', Env) -> K) = k(exp((E,E'),Env) -> + -> K) .
eq k(val(int(I),int(I')) -> + -> K) = k(val(int(I + I')) -> K) .
```

The semantics of the boolean expressions follows the same pattern as that of arithmetic expressions and the semantics of the conditional is immediate; we omit these here. The semantics of the assignment statement is now straightforward, because we have already defined the auxiliary operation `writeTo`:

```
eq k(exp(X := E, Env) -> K) = k(exp(E, Env) -> writeTo(Env[X]) -> val(nothing) -> K) .
```

Since the evaluation of any expression is expected to produce a value on the continuation and since `writeTo` takes a value and writes it in the store without placing any other value as a result on the continuation, we explicitly place the special value `nothing`[13] on the continuation as the value of the assignment; therefore, assignments return no values, they are only used for their side effects in the language. In fact, we use the value `nothing` as the result value of all language constructs that are intended to be statements. The corresponding type of `nothing` is `unit` (see Section 4.2).

The semantic definitions of sequential composition, blocks, loops and printing are explained in detail in [46]; we do not discuss them here. We next focus on the semantics of function calls. One can regard a function call as an abrupt change of control: the current control context is frozen, then the control is passed to the body of the function; if a `return` statement is encountered, then the frozen control context in which the function call took place is unfrozen and becomes the active one. Since function calls can be nested, the frozen control context needs to be stacked appropriately. This is the reason why we use the thread state attribute called `stack`. The semantic definition below should now be self-explanatory:

```
  op apply : Name -> ContinuationItem .
  op return : -> ContinuationItem .
  op freeze : Continuation -> ContinuationItem .
...
  eq k(exp(F(El), Env) -> K) = k(exp(El, Env) -> apply(F) -> K) .
  eq t(k(val(Vl) -> apply(F) -> K) stack(Stack) TS)
     globalEnv(Env) functions(Fs (function F(Xl) {local (LXl) ; E}))
   = t(k(val(Vl) -> bindTo((Xl,LXl), Env) -> exp*(E) -> return -> stop)
     stack(freeze(K) -> Stack) TS)
     globalEnv(Env) functions(Fs (function F(Xl) {local (LXl) ; E})) .
  eq k(exp(return(E), Env) -> K) = k(exp(E, Env) -> return -> K) .
  eq k(val(V) -> return -> K) stack(freeze(K') -> Stack) = k(val(V) -> K') stack(Stack) .
```

[12] Note the use of the built-in `if-then-else-fi` operator in the last equation. One could easily eliminate it by replacing that equation with two equations, one for "`val(bool(true))`" and one for "`val(bool(false))`".

[13] We do not declare this constant of sort `Value` here; see [46].

Let us next define the last and most complex feature of SIMPLE: threads. Creating a new thread is easy: all one needs to do is to add one more term of the form `t(...)` to the top level of the soup. The newly created term should encapsulate all the corresponding thread attributes. Note that we use a "stopping" continuation for the newly created threads, called `die`. The meaning of `die` is that threads simply die when they reach it:

```
  op lockv : Int -> Value .
  op die : -> Continuation .
  ops lock acquire release : -> ContinuationItem .
...
  var Is .               --- set of lock indexes (integers)
  var Cs : CounterSet .  --- pairs of the form [lock, counter]
  eq t(k(exp(spawn(E), Env) -> K) TS)
   = t(k(val(nothing) -> K) TS) t(k(exp(E, Env) -> die) stack(stop) holds(empty)) .
  eq t(k(val(V) -> die) holds(Cs) TS) busy(Is) = busy(Is - Cs) .
```

Threads without some mechanism for synchronization are close to useless. We chose one of the simplest mechanisms for SIMPLE, namely one based on locks. Since one would like to evaluate and possibly pass locks around just like any other values in the language, we add a new type of value to the language:

```
  eq k(exp(lock(E), Env) -> K) = k(exp(E, Env) -> lock -> K) .
  eq k(val(int(I)) -> lock -> K) = k(val(lockv(I)) -> K) .
```

A thread may acquire the same lock more than once; this situation typically appears when the statement of acquiring a lock is part of a recursive function, in such a way that each recursive function invocation results in acquiring the same lock. Before physically releasing a lock to the runtime environment, one should make sure that the thread requests releasing it as many times as it acquired that lock. This is the semantics of locking in most multithreaded languages, including JAVA. An important observation here is that, once a thread already holds a given lock, subsequent acquisitions of the same lock are purely local operations that cannot affect the execution of the other threads. Therefore, we can define subsequent lock acquiring using an equation rather than a rule. However, note that the first acquisition of the lock must be defined using a rule, whereas the release can be defined entirely with equations:

```
  eq k(exp(acquire(E), Env) -> K) = k(exp(E, Env) -> acquire -> K) .
  eq k(val(lockv(I)) -> acquire -> K) holds([I, N] Cs)
   = k(val(nothing) -> K) holds([I, N + 1] Cs) .
 crl t(k(val(lockv(I)) -> acquire -> K) holds(Cs) TS) busy(Is)
  => t(k(val(nothing) -> K) holds([I, 0] Cs) TS) busy(I Is)  if not(I in Is) .
  eq k(exp(release(E), Env) -> K) = k(exp(E, Env) -> release -> K) .
  eq k(val(lockv(I)) -> release -> K) holds([I, Nz] Cs)
   = k(val(nothing) -> K) holds([I, Nz - 1] Cs) .
  eq t(k(val(lockv(I)) -> release -> K) holds([I, 0] Cs) TS) busy(I Is)
   = t(k(val(nothing) -> K) holds(Cs) TS) busy(Is) .
```

Note that the semantics of lock release has been defined using only equations. That is because threads do not "compete" on the release of locks; in other words, there is no possible scenario in which a lock release may lead to a new behavior of the program. This is not the case for lock acquisition, because, in a scenario in which two threads can each acquire the same lock, depending on which one takes it we get potentially two different behaviors of the program. In general, the problem of deciding when to use equations and when to use rules is at least as difficult as saying when a specification is confluent (because what we need here is confluence of the equations only, and only on terms that result from executions of well-formed programs); therefore, unfortunately, one cannot hope to decide it automatically in all situations. The safe approach when one is not sure whether an equation should be a rule or not is to just make it a rule; the drawback of having too many rules is an increased state space, which can lead to a decrease in performance when one analyzes programs.

We have defined all the features that we want to include in our language. The only thing left to do is to put everything together. We do this by including all the modules defining the semantics of each of these features, as we did when we put all the syntax together, and then defining an `eval` operation on programs, whose result is a list of integers (the output generated with the `print` command): "`op eval : Pgm -> [IntList]`". Note that the `eval` operation above actually returns a kind. That is because a program may not always evaluate properly. For example, a program may not be well-typed (a type-checker could remove this worry), may terminate unexpectedly (division by zero), or may not terminate. We define the semantics of `eval` using an auxiliary operation which creates the appropriate initial state. The program terminates when its main thread terminates, that is, when a value is passed to the starting continuation, `stop` (`#` defines the length on lists and `locs(N)` reduces to the list of `N` locations):

```
...
  var Fs : FunctionSet .  var Il : IntList .
  var TS : SimpleThreadState .
  eq eval(Fs) = eval(global nil ; Fs) .
  op [_] : SimpleState -> [IntList] .
  eq eval(global Xl ; Fs)
   = [t(k(exp(main(), empty) -> stop) stack(stop) holds(empty))
      globalEnv(empty[Xl <- locs(#(Xl))]) nextLoc(#(Xl))
      mem(empty) output(nil) busy(empty) functions(Fs)] .
  eq [t(k(val(V) -> stop) TS) output(Il) S] = Il .
```

The semantics of SIMPLE is now complete. The first benefit one gets from this definition is an *interpreter for free*. Indeed, all one needs to do is to start a Maude rewrite session using the command "`rew eval(program)`", where `program` can be any program that parses. In Section 4 we show how one can use the exact same definition of SIMPLE to formally *analyze* programs in various ways.

3.2. Other language case studies

The SIMPLE language discussed in Section 3.1 illustrates a particular language specification, which is just one example within a much broader language specification methodology, based on a first-order representation of continuations. A key point worth making is that this methodology *scales up* quite well to real languages with complex features, both in terms of still allowing very readable and understandable specifications, and also in being capable of providing high performance interpreters and competitive program analysis tools.

For example, Java 1.4 (see also [13] for a complete formal semantics) and the JVM have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [27,25]. In fact, the semantics of large fragments of conventional languages are routinely developed by UIUC graduate students taking programming language design and semantics classes, as course or short research projects, including, besides Java and the JVM, languages like (alphabetically), Beta, Haskell, Lisp, LLVM, Pict, Python, Ruby, Scheme, and Smalltalk [56].

Typically, one needs two equations or rewrite rules to define the semantics of each language construct: one to *divide* the evaluation task into evaluation subtasks, and the other to *conquer* the task by combining the values produced by the evaluations of the subtasks into a resulting value for the original task. However, there are language constructs that can be translated into other, more general constructs with just one equation (e.g., for loops into while loops), but also language constructs that need many equations or rules. For example, the creation of a new object in Java needs more than 10 equations. Each equation defines a meaningful and different case to analyze, which cannot be avoided or collapsed as a special case of a more general case neither technically nor conceptually; this is due to the inherent complexity of object creation in the presence of inner classes.

A semantics of a Caml-like language with threads was discussed in detail in [45], and a modular rewriting logic semantics of a subset of CML has been given by Chalub and Braga in [12]. Following a continuation-based semantics similar to the one in this paper, D'Amorim and Roşu have given a definition of the Scheme language in [23]. Other language case studies, all specified in Maude, include BC [7], CCS [70,7], CIAO [66], Creol [37], ELOTOS [68], MSR [10,64], PLAN [65,66], the ABEL hardware description language [38], SILF [36], FUN [57], and the π-calculus [67]. Some of these rewrite logic language definitions do not obey the continuation-based style advocated in this paper. That is because those languages lack complex control statements, such as exceptions, or break/continue of loops, or abrupt return from functions, or halt, or call/cc, which the continuation-based style can handle naturally. Nevertheless, those languages can also be given a continuation-based semantics.

3.3. Towards a specialized notation and automatic interpreter generation

To ease the process of developing and understanding large rewrite logic definitions of programming languages, as well as to facilitate the automatic translation of such definitions into very efficient interpreters, we are currently investigating a domain-specific definitional framework within rewriting logic, called K, together with partially automated translations of K language definitions into rewriting logic and into C. The *K-notation* consists of a series of syntactic-sugar conventions, including ones for matching modulo axioms, for eliding unnecessary variables, for sort inference, and for context transformers. As part of our ongoing research, we are developing a number of tools around K to assist in defining and analyzing programming languages. We currently perform the translation of K

definitions into Maude or into C interpreters by hand following a mechanical procedure, with ongoing work on an automated translation. The technical report [57] presents K in detail, while [36] shows an instance of using it to define a language called SILF; here we only informally discuss it by means of a simple example. Consider the following Maude definition of a variable lookup operation in a conventional programming language, together with the afferent Maude variable declarations:

```
var X : Name .  var K : Continuation .
var L : Location .  var V : Value .
var Env : Environment .  var Mem : Store .
eq k(exp(X) -> K) env([X,L] Env) store([L,V] Mem)
 = k(val(V) -> K) env([X,L] Env) store([L,V] Mem)
```

First, note that variable declarations take almost as much space as the actual equation, while the sorts of *all* the variables can be easily inferred from the context: the sorts of X and L are `Name` and `Location`, because they appear together as a pair in an environment; similarly for the sort of V; then *Env* and *Mem* can have only sorts `Environment` and `Store`, respectively, because they appear as arguments of operations that were declared to take arguments of these sorts; finally, K can also be inferred to have sort `Continuation`, because it appears as argument of the continuation constructor operation. In K, we assume that the sorts of all variables can be automatically inferred from their use; note that users can tag terms, in particular variables, with sorts for clarity or to help the sort inference procedure.

Second, note that the particular name of the remaining continuation (K), environment (*Env*) and store (*Mem*) not only remain unchanged, but also are not even necessary to create the context; all we need to know is that, e.g., the pair `[X,L]` appears somewhere in the environment. Since this is also a very common situation in language definitions, in K we use left ("⟨") and right ("⟩") angle brackets instead of the usual parentheses whenever we want to state that the enclosed associative or associative and commutative list can have other elements to the left or to the right.

Third, note that most of the subterms appearing in this equation are there only to enforce a particular structure of the state in order for the equation to be applicable; for example, the subterms rooted in `env` and `store`. Consequently, the same subterms are simply repeated in the right-hand side, because we do not want them to change; this situation appears often in language definitions and it may be sometimes error prone, because one can forget one subterm or one can, by mistake, misspell the name of a variable. In K, we first write the context, then we underline *only* the subterms that change, and then write under the line the terms that should replace the underlined ones. With these notational conventions, the Maude definition of variable lookup can be written in K as a *contextual rule* simply as follows:

$$\frac{k(exp(X))\rangle\ env\langle[X, L]\rangle\ store\langle[L, V]\rangle.}{val(V)}$$

The translation of K definitions into rewriting logic enables the use of the various analysis tools developed for rewrite logic specifications, as shown in the next section, while the translation into C allows for very efficient interpreters. While Maude yields a relatively efficient interpreter for a language defined as a rewrite logic specification, such an interpreter results from a general purpose, interpreted rewrite engine. The translation of K to C can be, in some sense, regarded as a compiler for a *fragment* of rewriting logic, namely one which is relevant for programming language definitions. A suite of tests performed for SILF are discussed in [36] and essentially show that the C interpreter hand-generated (but mechanically) from SILF's rewrite logic definition is less than an order of magnitude slower than a standard, optimized interpreter for Java.

4. Program analysis techniques and tools

Specifying formally the rewriting logic semantics of a programming language in Maude yields a prototype interpreter for free. Thanks to generic analysis tools for rewriting logic specifications currently provided as part of the Maude system, we additionally get the following analysis tools also *for free*:

(1) a *semi-decision procedure* to find failures of safety properties in a (possibly infinite-state) concurrent program using Maude's `search` command;
(2) an LTL *model checker* for finite-state programs or program abstractions;
(3) a *theorem prover* (Maude's ITP [19,20]) that can be used to prove programs correct semi-automatically.

We discuss the first two items in Section 4.1, where we give some examples illustrating this kind of automated analysis for programs in SIMPLE. Analyses based on abstract semantics are discussed in Section 4.2, and deductive approaches are discussed in Section 4.3.

4.1. Search and model checking analysis

In this section we illustrate the search and model checking capabilities that one obtains for free from a rewrite logic semantic definition of a programming language. Let us consider again the definition of SIMPLE, together with the dining philosophers program below. If one executes that program using the command `rew eval(program)` then most likely one will see a normal execution, that is, one which terminates and outputs nothing. That is because there is a very small likelihood that the program will deadlock. Nevertheless, the potential for deadlock is there, meaning that some other executions of the same program may deadlock, with all the usual, undesired consequences.

To analyze all the possible rewriting computations from an initial state in a given rewriting logic specification, Maude provides a `search` command. This command takes an initial state to analyze, a pattern to be reached, and, optionally, a semantic condition to be satisfied by the reached pattern, and searches through all the state space generated in a breadth-first manner, by considering all the different rewrite rules that can be applied to each reachable state. Once one defines a rewriting logic specification of a language in Maude, one can simply use the built-in search capabilities of Maude to exhaustively search for executions of interest through the state space of a given program. The following search command generates all the states in which the dining philosophers program can deadlock:

```
search eval(                              function main() {
                                            local i ;
  global n ;                                n := 3 ;
                                            for(i := 1 ; i <' n ; i := i + 1)
  function f(x) {                             spawn(f(i)) ;
    acquire lock(x) ;                       acquire lock(n) ;
      acquire lock(x + 1) ;                   acquire lock(1) ;
        --- eat                                 --- eat
      release lock(x + 1) ;                   release lock(1) ;
    release lock(x)                         release lock(n)
  }                                       }

--- go to right column                                ) =>! Il:[IntList] .
```

The suffix `... =>! Il:[IntList]` tells the `search` command to search for all the normal forms of kind `[IntList]`, that is, all the normal forms of that program. As expected, the above returns *two* normal forms, namely one in which the program terminates and one in which each thread acquired one lock and is waiting, in a deadlock, for the other one to be released:

```
Solution 1 (state 361)
states: 485  rewrites: 8722 in 76ms cpu (77ms real) (113290 rewrites/second)
Il:[ExpList,IntSet,FindResult] --> (nil).List

Solution 2 (state 1140)
states: 1232  rewrites: 25741 in 240ms cpu (241ms real) (106825 rewrites/second)
Il:[ExpList,IntSet,FindResult] --> [t(k(val lockv(1) -> acquire -> discard -> exp((release
lock(1) ; release lock(n)), [i,loc(1)][n,loc(0)]) -> return -> stop) stack(freeze(stop) ->
stop) holds([3,0])) t(k(val lockv(2) -> acquire -> discard -> exp((release lock(x + 1) ;
release lock(x)), [n,loc(0)] [x,loc(2)]) -> return -> stop) stack(freeze(die) -> stop)
holds([1,0])) t(k(val lockv(3) -> acquire -> discard -> exp((release lock(x + 1) ;
release lock(x)), [n,loc(0)] [x,loc(3)]) -> return -> stop) stack(freeze(die) -> stop)
holds([2,0])) nextLoc(4) mem([loc(0),int(3)] [loc(1),int(3)] [loc(2),int(1)] [loc(3),int(2)])
output nil globalEnv([n,loc(0)]) busy(1 # 2 # 3) functions (function f x {local nil ;
acquire lock(x) ; acquire lock(x + 1) ; release lock(x + 1) ; release lock(x)}
function main nil {local i ; n := 3 ; i := 1 ; while i <' n (spawn f i ; i := i + 1) ;
acquire lock(n) ; acquire lock(1) ; release lock(1) ; release lock(n)})]

No more solutions.
states: 1406  rewrites: 30078 in 283ms cpu (284ms real) (105924 rewrites/second)
```

A common fix for the dining philosophers deadlock is to force the philosophers to follow a certain discipline in acquiring the forks: philosophers on odd positions acquire the left fork first and then the right one, while philosophers on even positions take the right fork first followed by the second. As expected, if one fixes the code above, then the search command returns only one solution, the one reflecting a normal termination of the concurrent program. However, the above-mentioned deadlock is not the only program flaw. Consider the slightly modified version of the

deadlock-free version of the program above, where each philosopher continues to alternatively think and eat forever. A property worth checking for this program is to see whether a certain philosopher (say philosopher 3) starves or not. To check this, it suffices to define a parametric predicate `eaten(i)` which holds in the state where philosopher `i` is eating. Then, using Maude's built-in LTL model checker, one can simply check whether the LTL formula `[]<>eaten(i)` holds or not as follows:

```
red modelCheck(eval(
  global n ;
  function f(x) {
    while(true) {
      if x % 2 ==' 1
      then {
        acquire lock(x) ;
          acquire lock(x + 1) ;
            --- eat
          release lock(x + 1) ;
        release lock(x)
      }
      else {
        acquire lock(x + 1) ;
          acquire lock(x) ;
            --- eat
          release lock(x) ;
        release lock(x + 1)
      }
      --- think
    }
  }
--- go to right column
function main() {
  local i ;
  n := 3 ;
  for(i := 1 ; i <' n ; i := i + 1)
    spawn(f(i)) ;
  while (true) {
    if n % 2 ==' 1
    then {
      acquire lock(n) ;
        acquire lock(1) ;
          --- eat
        release lock(1) ;
      release lock(n)
    }
    else {
      acquire lock(1) ;
        acquire lock(n) ;
          --- eat
        release lock(n) ;
      release lock(1)
    }
  }
  --- think
}
), []<> eaten(3)) .
```

which, as one expects, returns a counterexample in which philosophers 1 and 2 keep eating alternatively, and philosopher 3 never gets a chance to eat.

It is well-known that concurrency leads to massive increases in the state space of a program, because there are very many equivalent interleavings of the same computation that have to be checked by a standard model checker. One way to avoid this state explosion is to use *partial order reduction* (POR) techniques, (see [16] and references there), in which many of these interleaving computations are never explored. POR is *complete*, in the sense that an LTL formula not involving the "next" operator $\bigcirc$ can be shown to hold using POR model checking iff it can be shown to hold using standard model checking [16]. The traditional way to provide a POR model checking capability for a given programming language $\mathcal{L}$ is to *modify the model checking algorithm* of a model checker for $\mathcal{L}$. This is a substantial task, which furthermore has to be performed for each different language. Since we are interested in amortizing the cost of *all* program analysis tools across many languages by making them generic, A. Farzan and the first author have developed a POR model checking technique [26] that is generic in the language $\mathcal{L}$, under very general assumptions about $\mathcal{L}$, such as having processes or threads endowed with unique identities. An important advantage of this language-generic POR technique is that it does not require any changes to an underlying model checker. In particular, it can be used together with the Maude LTL model checker to model check with POR programs in any programming language $\mathcal{L}$ satisfying a few general assumptions. The key idea is to perform a *theory transformation* of the rewrite theory $\mathcal{R}_{\mathcal{L}}$ specifying the semantics of $\mathcal{L}$ to obtain a POR-enabled, semantically equivalent rewrite theory $\mathcal{R}_{\mathcal{L}}^{por}$. We can then use a standard LTL model checker to model check programs in $\mathcal{L}$ with POR reduction by model checking them in a standard way using $\mathcal{R}_{\mathcal{L}}^{por}$. Experiments with semantic definitions for the JVM and a Promela-like language suggest that the ratios of state-space reduction obtained with this generic POR technique are comparable to those reported using language-specific model checkers with a built-in POR capability [26].

4.2. Analyses based on abstract semantics

The three types of analyses discussed so far, namely interpretation/simulation, search and model checking, make use of the semantic rewriting logic definition of a programming language *as is*. Therefore, a language designer obtains all these analysis capabilities essentially *for free*. There are, however, certain kinds of analyses that require a slightly different, typically more abstract semantics to be defined. One should *not* regard the need for a different semantics as a breach of modularity, but rather as defining a totally different system, or "language", namely one that "interprets"

the syntax differently. Interestingly, one can do this relatively easily, by just modifying the existing language semantic definitions appropriately.

The already existing semantic definition of the language acts as a check-list, telling the analysis tool developer *what* needs to be defined and only partly *how* to define it. The tool developer is responsible for filling in all the details. In the case of simple analyzers, such as a type checker or an abstract interpreter in which the abstract domain and its properties can be inferred from the concrete domain in some straightforward manner, one can envision automatic generators of analysis tools, by providing some general rules stating how the concrete semantics needs to be changed into an abstract one. While this is clearly an interesting research topic, we do not pursue it here. We assume that the tool developer is responsible for the entire definition of the analyzer. In this section we briefly discuss two kinds of static analysis tools that we have experimented with, namely type checkers and domain-specific certifiers.

Let us first elaborate on some intuitions underlying the definition of a type checker. To keep the discussion focused, let us assume a type checker for SIMPLE. Since a type checker is not concerned with the concrete values handled by a program, but instead with their types, we replace values in the definition of SIMPLE by types. The continuation item `val(...)` becomes `type(...)` and several constant types need to be added, such as `int`, `bool`, etc. Recall the continuation-based definition of comparison:

```
eq k(exp(E >’ E’, Env) -> K) = k(exp((E,E’), Env) -> > -> K) .
eq k(val(int(I),int(I’)) -> > -> K) = k(val(bool(I > I’)) -> K) .
```

Viewed through the prism of types, the above says that `E >’ E’` has the type `bool` if `E` and `E’` have the type `int`. It is then straightforward to modify the above equations as follows:

```
eq k(exp(E >’ E’, Env) -> K) = k(exp((E,E’), Env) -> > -> K) .
eq k(type(int,int) -> > -> K) = k(type(bool) -> K) .
```

Of course, environments in the concrete semantics become type environments in the abstract semantics, assigning types to names. One can modify the semantics of each language construct as above, thereby easily obtaining a type checker. However, one should carefully rethink the new (and more abstract) semantics of each construct carefully, using the previous semantics of the language only as a check-list of features to be redefined, because not all constructs have always a mechanical translation. For example, statements that change the control flow of a program may type-check differently from how they evaluate; for example, the conditional has the following type semantics:

```
eq k(exp(if BE then E else E’, Env) -> K) = k(exp((BE,E,E’), Env) -> if -> K) .
eq k(type(bool,T,T) -> if -> K) = k(type(T) -> K) .
```

The above type policy allows the conditional to be used also in non-statement contexts (like `_?_:_` in Java and C++); if one wants to enforce a stricter type policy for conditional, then one can replace `T` by `unit` (corresponding to “statements”; this is also the type of the assignment and of loops).

We defined several type checkers following this semantic abstraction methodology as part of our programming language courses [56]. Students also developed such type checkers as homework assignments, including ones based on type inference. In the case of type reconstruction, the result of “evaluating” an expression is a set of equational type constraints. All these type constraints are solved either at the end of the evaluation process or on the fly.

Another category of analysis tools that we investigated, also derived from the semantics of the programming language, is that of domain-specific certifiers. Like in type checking, expressions evaluate to some abstract values. However, unlike in type checking, these abstract values have no relationship whatsoever with the concrete values. The abstract values make sense only in the context of a specific domain of interest, which also needs to be formally defined. Consider, for example, the domain of units of measurement, which can be formalized as an abelian group generated by the basic units (meter, second, foot, etc.)—suppose that multiplication of units is written as concatenation. A program certifier for this domain would check that, in a program written in an extended syntax allowing annotations specifying the units of variables, all the operations performed by the given program are consistent with the intuitions of the domain of units of measurement. For example, only expressions which have the same unit can be added or compared, while expressions of any units can be multiplied. The semantic definitions of addition and multiplication in this domain-specific certifier for SIMPLE would be:

```
eq k(exp(E + E’, Env) -> K) = k(exp((E,E’), Env) -> + -> K) .
eq k(unit(U,U) -> + -> K) = k(unit(U) -> K) .

eq k(exp(E * E’, Env) -> K) = k(exp((E,E’), Env) -> * -> K) .
eq k(unit(U,U’) -> * -> K) = k(unit(U U’) -> K) .
```

Formal definitions of domain-specific certifiers built on rewriting logic semantic programming language definitions have been investigated in depth in several places. In [15,14] we discuss such certifiers for the domains of units of measurement and a large fragment of C, in [39] we present a domain-specific certifier for the domain of coordinate frames, and in [58] one for the domain of optimal state estimation.

Each analysis tool has its particularities and may raise complex issues, from difficulty in defining it to intractability. The main point we want to stress in this section is that the original rewriting semantics of the programming language gives us a very useful skeleton on which to develop potentially any desired program analysis tool.

4.3. Logics of programs and semantics-based theorem proving

Given a programming language $\mathcal{L}$, we are often interested in using a *logic of programs* for $\mathcal{L}$ to reason about programs in $\mathcal{L}$. Two important tasks appear in this regard:

(1) the correctness of the chosen logic of programs for the given language $\mathcal{L}$ has to be justified in term of a mathematical definition of $\mathcal{L}$'s semantics; and
(2) *mechanizing* a logic of programs for $\mathcal{L}$ typically requires not only mechanizing the logic's inference rules, but also the discharging of *verification conditions* (VCs) generated by the inference process; and for discharging such VCs one often needs to use properties of $\mathcal{L}$'s underlying semantics.

Having a mathematically precise semantics of a programming language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be very useful for tasks (1) and (2). An important case study for task (1), showing the usefulness of the specification $\mathcal{R}_{\mathcal{L}}$ when $\mathcal{L}$ is Java source code, has been recently carried out by Ahrendt, Roth, and Sasse [59,1]. The goal was to validate automatically the correctness of a substantial subset (about 50 inference rules) of the JavaCard Dynamic logic [3], which has inference rules, implemented by so-called *taclets*, reducing the proof of a dynamic logic formula to that of simpler such formulas. Of particular interest in this case study were the *code transformation taclets*, that transform a program p to a simpler equivalent form p'. Of course, p, p' typically are not concrete JavaCard programs; they are instead *patterns*, symbolic expressions called program schemes. In the case of program schemes, the axioms in $\mathcal{R}_{Java}$ are insufficient to reason about semantic equivalence between the program schemes p and p' in a taclet. The elegant solution adopted in [59,1] is to lift $\mathcal{R}_{Java}$ to the symbolic level by specifying a more expressive semantics $\mathcal{R}_{Java}^{lift}$. Using the lifted semantics $\mathcal{R}_{Java}^{lift}$, over 50 code transformation taclets have been automatically validated using Maude. Furthermore, all axiomatic propositional logic taclets have also been automatically validated in Maude [59,1].

The work of Garrido, Meseguer, and Johnson on correctness of Cpp refactorings [30] also focuses on task (1). It provides, to the best of our knowledge, the first formal semantics of the C Preprocessor language (Cpp), and of program refactorings that change Cpp code. Refactoring of C code is not effective in practice if it does not support Cpp refactoring and does not preserve the modular structure allowed by Cpp, where a C program is made up of a directory of files with Cpp directives on how to assemble them together. The work in [30] gives a formal semantics in Maude of Cpp and of several Cpp refactorings, and then uses the formal Cpp semantics to give a mathematical proof of correctness of those Cpp refactorings. This serves as a foundation for the CRefactory tool [29], where such refactorings have been implemented.

Another substantial case study, this time involving both tasks (1) and (2), has centered on the Pascal-like language used in [31]. A Hoare logic for a substantial fragment of this Pascal-like language has been given in [43], where the correctness of the Hoare rules is mathematically justified on the basis of the language's formal semantics $\mathcal{R}_{\mathcal{L}}$, thus addressing task (1). Task (2) has been addressed by Clavel and Santa-Cruz in [21] using the Maude inductive theorem prover (ITP) as the underlying proof engine. Their ASIP + ITP tool integrates the theory $\mathcal{R}_{\mathcal{L}}$ with the Maude ITP and directly supports some Hoare rules. A user can state goals as Hoare triples; the Hoare rules are then applied by the ASIP tool to generate VCs, which can be discharged using the ITP [21].

In the same vein as ASIP + ITP, but going considerably further, the work on Java + ITP [60] by Sasse and Meseguer also addresses tasks (1) and (2). The paper [60] describes: (i) a Hoare logic for a subset of sequential Java; (ii) a mathematical proof of correctness of those Hoare rules based on the Maude semantics for the language, which is essentially an equational subset of the more general continuation-passing rewriting semantics for Java in [25]; (iii) the mechanization of this Hoare logic in Maude, so that Hoare triples for Java programs in this subset can be decomposed into smaller triples using the Hoare rules; and (iv) the machine-assisted discharging of the first-order verification conditions associated to Hoare triples by inductive reasoning, based on the Maude Java semantics and

using the Maude ITP. Java + ITP is used not only for experimentation, but also for teaching purposes in several graduate courses at UIUC.

Another relevant case study involves the Hennessy–Milner logic of programs for CCS [70,68]. Since the justification of the Hennessy–Milner logic is well-established, this work focuses on task (2). A. Verdejo and N. Martí-Oliet first give a rewriting logic semantics for CCS as a rewrite theory $\mathcal{R}_{CCS}$ in Maude. Then, an inference system for the Hennessy–Milner modal logic of CCS is also defined in Maude as another module at the meta-level that imports the module specifying $\mathcal{R}_{CCS}$ at the object level. In this way, the CCS interpreter obtained by the Maude specification of $\mathcal{R}_{CCS}$ is seamlessly extended into a program reasoning tool for CCS, in which the satisfaction of a Hennessy–Milner logic formula ϕ by a finitary CCS process P can be automatically verified by the Maude-based tool [70,68].

5. Conclusions and future directions

We have explained how rewriting logic can be used as a framework to unify equational semantics and SOS; and how, using a language such as Maude and its generic tools, efficient interpreters and analysis tools can be generated from language definitions. This paper is just a snapshot of what we believe is a promising collective research project. Much work remains ahead. We list below some future research directions that we find particularly attractive:

Modularity. A fully modular definitional style for rewriting logic has already been developed in [44]. An interesting open question is: what other definitional styles can likewise be endowed with a fully modular methodology? At the experimental level this should lead to a well-crafted library of modular semantic definitions in the spirit of MSOS, so that new language definitions can easily be developed by composing the semantic definitions of their basic features, changing their generic abstract syntax to the concrete syntax of the language in question.

Semantic equivalence and compiler generation. It would be highly desirable to develop general methods to show that two semantic definitions of a programming language are equivalent. Meta-results of this kind could be the basis of automated semantics-preserving translations between language definitions given in different definitional styles. They could also be the basis of generic formal compiler techniques; and of compiler generators that take a formal language definition as input and are provably correct, in the sense of preserving the language's semantics.

Generic tools. Although some quite useful generic tools already exist, it is clear that much more can be done. For example, it would be quite useful to have a generic *abstraction tool*, so that an infinite-state program in any language satisfying minimal requirements can be model checked by model checking a finite-state abstraction. Similarly, a *language-generic theorem proving tool* allowing the kind of reasoning supported at present by language-specific tools such as ASIP + ITP [21] and Java + ITP [60] for a large class of languages would likewise be highly desirable.

Acknowledgements

We cordially thank Wolfgang Ahrendt, Christiano Braga, Feng Chen, Manuel Clavel, Marcelo D'Amorim, Santiago Escobar, Azadeh Farzan, Mark Hills, Narciso Martí-Oliet, Peter Mosses, Andreas Roth, Ralph Sasse, Mark-Oliver Stehr, Traian Florin Şerbănuţă, Carolyn Talcott, Alberto Verdejo, and the UIUC students attending the programming language courses [56], who defined many real programming languages, for their various kinds of help with this paper, including examples, careful comments, and their own intellectual contributions to the rewriting logic semantics program. This work has been partially supported by the following grants: ONR N00014-02-1-0715, NSF/NASA CCF-0234524, NSF CAREER CCF-0448501, and NSF CNS-0509321.

References

[1] W. Ahrendt, A. Roth, R. Sasse, Automatic validation of transformation rules for Java verification against a rewriting semantics, in: G. Sutcliffe, A. Voronkov (Eds.), Proceedings of LPAR'05, in: LNCS, vol. 3835, Springer, 2005, pp. 412–426.

[2] D. Basin, G. Denker, Maude versus Haskell: An experimental comparison in security protocol analysis, in: K. Futatsugi (Ed.), Proceedings of WRLA'00, in: ENTCS, vol. 36, Elsevier, 2000.

[3] B. Beckert, A dynamic logic for the formal verification of Java Card programs, in: I. Attali, T. Jensen (Eds.), Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, in: LNCS, vol. 2041, Springer, 2001, pp. 6–24.

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

[5] A. Bouhoula, J.-P. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, Theoretical Computer Science 236 (2000) 35–132.

[6] C. Braga, Rewriting logic as a semantic framework for modular structural operational semantics, Ph.D. Thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[7] C. Braga, J. Meseguer, Modular rewriting semantics in practice, in: Proceedings of WRLA'04, in: ENTCS, vol. 117, Elsevier, 2004.
[8] M. Broy, M. Wirsing, P. Pepper, On the algebraic definition of programming languages, ACM Transactions on Programming Languages and Systems 9 (1) (1987) 54–99.
[9] R. Bruni, J. Meseguer, Generalized rewrite theories, in: Proceedings of ICALP'03, in: LNCS, vol. 2719, 2003, pp. 252–266.
[10] I. Cervesato, M.-O. Stehr, Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types, in: P. Degano (Ed.), Proceedings of WRLA'04, in: ENTCS, vol. 117, Elsevier, 2004.
[11] F. Chalub, An implementation of modular SOS in Maude, Master's Thesis, Universidade Federal Fluminense, May 2005, http://www.ic.uff.br/~frosario/dissertation.pdf.
[12] F. Chalub, C. Braga, A modular rewriting semantics for CML, Journal of Universal Computer Science 10 (7) (2004) 789–807. http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics.
[13] F. Chen, G. Roşu, Rewriting logic semantics of Java 1.4, http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java.
[14] F. Chen, G. Roşu, Certifying measurement unit safety policy, in: Proceedings of ASE'03, IEEE, 2003, pp. 304–309.
[15] F. Chen, G. Roşu, R.P. Venkatesan, Rule-based analysis of dimensional safety, in: Proceedings of RTA'03, in: LNCS, vol. 2706, Springer, 2003, pp. 197–207.
[16] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 2001.
[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. Quesada, Maude: Specification and programming in rewriting logic, Theoretical Computer Science 285 (2002) 187–243.
[18] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Maude 2.0 Manual, June 2003, http://maude.cs.uiuc.edu.
[19] M. Clavel, F. Durán, S. Eker, J. Meseguer, Building equational proving tools by reflection in rewriting logic, in: CAFE: An Industrial-Strength Algebraic Formal Method, Elsevier, 2000. http://maude.cs.uiuc.edu.
[20] M. Clavel, M. Palomino, The ITP tool's manual. Universidad Complutense, Madrid, April 2005, http://maude.sip.ucm.es/itp/.
[21] M. Clavel, J. Santa-Cruz, ASIP + ITP: A verification tool based on algebraic semantics, in: Proceedings of PROLE'05, 2005. http://maude.sip.ucm.es/itp/asip/.
[22] D. Clément, J. Despeyroux, L. Hascoet, G. Kahn, Natural semantics on the computer, in: K. Fuchi, M. Nivat (Eds.), Proceedings, France–Japan AI and CS Symposium, ICOT, 1986, pp. 49–89. Also, Information Processing Society of Japan, Technical Memorandum PL-86-6.
[23] M. d'Amorim, G. Roşu, An equational specification for the scheme language, Journal of Universal Computer Science 11 (7) (2005) 1327–1348.
[24] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, in: F. Gadducci, U. Montanari (Eds.), Proceedings of WRLA'02, in: ENTCS, vol. 117, Elsevier, 2002.
[25] A. Farzan, F. Chen, J. Meseguer, G. Roşu, Formal analysis of Java programs in JavaFAN, in: Proceedings of CAV'04, in: LNCS, vol. 3114, Springer, 2004, pp. 501–505.
[26] A. Farzan, J. Meseguer, Making partial order reduction tools language-independent, in: G. Denker, C. Talcott (Eds.), Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications, in: ENTCS, Elsevier, 2006 (in press).
[27] A. Farzan, J. Meseguer, G. Roşu, Formal JVM code analysis in JavaFAN, in: Proceedings of AMAST'04, in: LNCS, vol. 3116, 2004, pp. 132–147.
[28] M. Felleisen, D.P. Freidman, Control operators, the SECD machine, and the λ-calculus, in: Formal Description of Programming Concepts III, Proceedings of IFIP TC2 Working Conference, North Holland, 1987, pp. 193–217.
[29] A. Garrido, Program refactoring in the presence of preprocessor directives, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 2005.
[30] A. Garrido, J. Meseguer, R. Johnson, Algebraic semantics of the C preprocessor and correctness of its refactorings, Technical Report UIUCDCS-R-2006-2688, University of Illinois at Urbana-Champaign, February 2006.
[31] J.A. Goguen, G. Malcolm, Algebraic Semantics of Imperative Programs, MIT Press, 1996.
[32] J.A. Goguen, K. Parsaye-Ghomi, Algebraic denotational semantics using parameterized abstract modules, in: J. Diaz, I. Ramos (Eds.), Formalizing Programming Concepts, in: LNCS, vol. 107, Springer, 1981, pp. 292–309.
[33] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J. Wright, Initial algebra semantics and continuous algebras, Journal of the Association for Computing Machinery 24 (1) (1977) 68–95.
[34] P.H. Hartel, LETOS—a lightweight execution tool for operational semantics, Software: Practice and Experience 29 (1999) 1379–1416.
[35] M. Hennessy, The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics, John Wiley & Sons, 1990.
[36] M. Hills, T.F. Şerbănuţă, G. Roşu, A rewrite framework for language definitions and for generation of efficient interpreters, in: Proceedings of WRLA'06, in: ENTCS, Elsevier, 2006 (in press).
[37] E.B. Johnsen, O. Owe, E.W. Axelsen, A runtime environment for concurrent objects with asynchronous method calls, in: N. Martí-Oliet (Ed.), Proceedings of WRLA'04, in: ENTCS, vol. 117, Elsevier, 2004.
[38] M. Katelman, J. Meseguer, A rewriting semantics for abel with applications to hardware/software co-design and analysis, in: G. Denker, C. Talcott (Eds.), Proceedings of WRLA'06, in: ENTCS, Elsevier, 2006 (in press).
[39] M. Lowry, T. Pressburger, G. Roşu, Certifying domain-specific policies, in: Proceedings of ASE'01, IEEE, 2001, pp. 81–90.
[40] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1) (1992) 73–155.
[41] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce (Ed.), Proceedings of WADT'97, in: LNCS, vol. 1376, Springer, 1998, pp. 18–61.
[42] J. Meseguer, Software specification and verification in rewriting logic, in: M. Broy, M. Pizka (Eds.), Models, Algebras, and Logic of Engineering Software, NATO Advanced Study Institute, Marktoberdorf, Germany, July 30–August 11, 2002, IOS Press, 2003, pp. 133–193.
[43] J. Meseguer, Lecture notes on program verification. CS 476, University of Illinois, Spring 2005, http://www.cs.uiuc.edu/class/fa05/cs476/.

[44] J. Meseguer, C. Braga, Modular rewriting semantics of programming languages, in: Proceedings of AMAST'04, in: LNCS, vol. 3116, Springer, 2004, pp. 364–378.
[45] J. Meseguer, G. Roşu, Rewriting logic semantics: From language specifications to formal analysis tools, in: Proceedings of IJCAR'04, in: LNAI, vol. 3097, Springer, 2004, pp. 1–44.
[46] J. Meseguer, G. Roşu, The rewriting logic semantics project, Technical Report UIUCDCS-R-2005-2639, University of Illinois at Urbana-Champaign, 2005. Ask authors for the complete Maude code.
[47] R. Milner, Functions as processes, Mathematical Structures in Computer Science 2 (2) (1992) 119–141.
[48] J. Moore, R. Krug, H. Liu, G. Porter, Formal models of Java at the JVM level—a survey from the ACL2 perspective, in: Proc. Workshop on Formal Techniques for Java Programs, in Association with ECOOP 2001, 2002.
[49] P.D. Mosses, Unified algebras and action semantics, in: Proceedings of STACS'89, in: LNCS, vol. 349, Springer, 1989, pp. 17–35.
[50] P.D. Mosses, Denotational semantics, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, North-Holland, 1990, (Chapter 11).
[51] P.D. Mosses, Foundations of modular SOS, in: Proceedings of MFCS'99, in: LNCS, vol. 1672, Springer, 1999, pp. 70–80.
[52] P.D. Mosses, Pragmatics of modular SOS, in: Proceedings of AMAST'02, in: LNCS, vol. 2422, Springer, 2002, pp. 21–40.
[53] P.D. Mosses, Modular structural operational semantics, Journal of Logic and Algebraic Programming 60–61 (2004) 195–228.
[54] M. Pettersson, Compiling Natural Semantics, in: LNCS, vol. 1549, Springer, 1999.
[55] G.D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60–61 (2004) 17–139. Previously published as technical report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
[56] G. Roşu, Programming language design and semantics classes, Department of Computer Science, University of Illinois at Urbana-Champaign. http://fsl.cs.uiuc.edu/~grosu/classes/.
[57] G. Roşu, K: A Rewrite Logic Framework for Language Design, Semantics, Analysis and Implementation, Technical Report UIUCDCS-R-2005-2672, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.
[58] G. Roşu, R.P. Venkatesan, J. Whittle, L. Leuştean, Certifying optimality of state estimation programs, in: Proceedings of CAV'03, in: LNCS, vol. 2725, Springer, 2003, pp. 301–314.
[59] R. Sasse, Taclets vs. rewriting logic—relating semantics of Java, Master's Thesis, Fakultät für Informatik, Universität Karlsruhe, Technical Report in Computing Science No. 2005-16, Germany, May 2005, http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2005/16.
[60] R. Sasse, J. Meseguer, Java + ITP: A verification tool based on Hoare logic and algebraic semantics, in: G. Denker, C. Talcott (Eds), Proceedings of WRLA'06, in: ENTCS, Elsevier, 2006 (in press).
[61] D.A. Schmidt, Denotational Semantics—A Methodology for Language Development, Allyn and Bacon, Boston, MA, 1986.
[62] D. Scott, 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.
[63] D. Scott, C. Strachey, Toward a mathematical semantics for computer languages, in: Proc. Symp. on Comp. and Automata, in: Microwave Research Institute Symposia Series, vol. 21, Polytechnical Institute of Brooklyn, 1971.
[64] M.-O. Stehr, I. Cervesato, S. Reich, An execution environment for the MSR cryptoprotocol specification language, http://formal.cs.uiuc.edu/stehr/msr.html.
[65] M.-O. Stehr, C. Talcott, PLAN in Maude: Specifying an active network programming language, in: F. Gadducci, U. Montanari (Eds.), Proceedings of WRLA'02, in: ENTCS, vol. 117, Elsevier, 2002.
[66] M.-O. Stehr, C.L. Talcott, Practical techniques for language design and prototyping, in: J.L. Fiadeiro, U. Montanari, M. Wirsing (Eds.), Abstracts Collection of the Dagstuhl Seminar 05081 on Foundations of Global Computing, February 20–25, 2005, Schloss Dagstuhl, Wadern, Germany, 2005.
[67] P. Thati, K. Sen, N. Martí-Oliet, An executable specification of asynchronous Pi-calculus semantics and may testing in Maude 2.0, in: F. Gadducci, U. Montanari (Eds.), Proceedings of WRLA'02, in: ENTCS, vol. 117, Elsevier, 2002.
[68] A. Verdejo, Maude como marco semántico ejecutable, Ph.D. Thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.
[69] A. Verdejo, N. Martí-Oliet, Executable structural operational semantics in Maude, Manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.
[70] A. Verdejo, N. Martí-Oliet, Implementing CCS in Maude 2, in: F. Gadducci, U. Montanari (Eds.), Proceedings of WRLA'02, in: ENTCS, vol. 117, Elsevier, 2002.
[71] P. Viry, Equational rules for rewriting logic, Theoretical Computer Science 285 (2002) 487–517.
[72] M. Wand, First-order identities as a defining language, Acta Informatica 14 (1980) 337–357.