SOS'05, ENTCS 156, pp. 27-56. 2006
SOS 2005 Preliminary Version

The Rewriting Logic Semantics Project

José Meseguer and Grigore Roşu

{*meseguer,grosu*}*@cs.uiuc.edu*
Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Abstract

Rewriting logic is a flexible and expressive logical framework that unifies denotational semantics and SOS in a novel way, avoiding their respective limitations and allowing very succinct semantic definitions. The fact that a rewrite theory's axioms include both equations and rewrite rules provides a very useful "abstraction knob" to find the right balance between abstraction and 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.

Keywords. Semantics and analysis of programming languages, rewriting logic.

1 Introduction

The fact that rewriting logic specifications [32,8] provide 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 [58,5,59,55,53,34,56,13,47,57,23,21,30,6,35,36,11,10,22,17,48,1,54,19]. 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 global snapshot of this project – emphasizing the fact that one can obtain quite efficient interpreters and program analysis tools from the semantic definitions essentially *for free* – was given in [36]. But this is a fast-moving area, so that new developments and the opportunity of discussing aspects less emphasized in [36] make it worthwhile for us to attempt giving here a second snapshot. In our view, what makes this project promising is the combination of three interlocking facts:

(i) 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 denotational semantics and SOS in a novel way, avoiding their respective limitations and allowing very succinct semantic definitions;

(ii) that rewriting logic semantic definitions are *directly executable* in a rewriting logic language such as Maude [14], and can thus become quite efficient interpreters; and

(iii) that *generic formal tools* such as the Maude LTL model checker [20], the Maude inductive theorem prover [15,16], and new tools under development such as a language-generic partial order reduction tool [22], 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 necessarily imply inefficiency*: in some cases the analyses so obtained outperform those of well-known language-specific tools [23,21].

1.1 Semantics: Equational vs. SOS

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

In *equational semantics*, formal definitions take the form of *semantic equations*, typically satisfying the *Church-Rosser* property. Both higher-order and first-order versions have been shown to be useful formalisms. There is a vast literature in these areas that we do not attempt to survey. However, we can mention some early denotational semantics papers such as [50,51] and the surveys [49,40]. Similarly, we mention [61,27,7] for early algebraic semantics papers, some of which use initial algebra semantics, and [26] for a recent book. We use the more neutral term *equational semantics* to emphasize the fact that (higher-order) denotational and (first-order) algebraic semantics have many common features and can both be viewed as instances of a common equational framework. Strong points of equational semantics include: (1) it has a *model-theoretic*, denotational semantics given by *domains* in the higher-order case, and typically by *initial algebras* in the first-order case; (2) it has also a *proof-theoretic*, operational semantics given by *equational reduction* with the semantic equations; (3) semantic definitions can be easily turned into efficient interpreters, thanks to efficient higher-order functional languages (ML, Haskell, etc.) and first-order equational languages (ACL2, OBJ, ASF+SDF, etc.); (4) there is good higher-order and first-order theorem proving support.

However, equational semantics has the following drawbacks: (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*, unless the concurrency is that of a purely deterministic computation; (2) one can *indirectly model*[1] some concurrency

[1] Two good examples of indirectly modeling concurrency within a purely functional frame-

aspects with devices such as a scheduler, or lazy data structures, but a direct comprehensive modeling of all concurrency aspects remains elusive within an equational framework; (3) 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 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 [45]; there is again a vast literature on the topic that we do not attempt to survey; for a good textbook introduction see [29]. Strong points of SOS include: (1) it is a general, yet quite intuitive formalism, allowing detailed *step-by-step* modeling of program execution; (2) it has a simple *proof-theoretic* semantics using semantic rules as inference rules; (3) it is fairly well suited to model *concurrent languages*, 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 [41,42,43]; (3) although some tools have been built to execute SOS definitions (see for example [18,28,44]), tool support for verifying properties is perhaps less developed than for equational semantics.

1.2 Unifying SOS and Equational Semantics: the Abstraction Knob

For the most part, equational 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 clearly superior and tends to prevail as the formalism of choice, but for deterministic languages equational approaches are also widely used. Of course there are also practical considerations of tool support for both execution and formal reasoning.

In the end, equational semantics and SOS, although each very valuable

work are the ACL2 semantics of the JVM using a scheduler [39], and the use of lazy data structures in Haskell to analyze cryptographic protocols [2].

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 [32,8] does indeed provide one such unifying framework. The key to this, indeed very simple, unification is what we call rewriting logic's *abstraction knob*. The point is that in equational semantics' model-theoretic approach entities are *identified by the semantic equations*, and have unique *abstract denotations* in the corresponding models. In our knob metaphor this means that in equational semantics the abstraction knob 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 knob is *always turned down to its minimum position*.

How is the unification and corresponding availability of an abstraction knob 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 knob 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 knob is turned down to the minimum position.

Rewriting logic's "abstraction knob" 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 knob, 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. What can we do in general to make a specification *as abstract as possible*, that is, to "turn the knob up" as much as possible? We can identify a subset $R_0 \subseteq R$ such that: (1) $R_0 \cup E$ is Church-Rosser; and (2) R_0 is 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 seman-

[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 $(\Sigma, E, \mu, R, \phi)$.

tic equations for most of the sequential features of a programming language: only when interactions with memory could lead to nondeterminism (particularly if the language has threads, or they could later be added to the language in an extension) or for intrinsically concurrent features are rules (as opposed to equations) really needed.

The conceptual distinction between equations and rules 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 infeasible 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. 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 knob" to the right observability level by *converting some equations into rules*. That is, we can regulate the knob to find for each kind of analysis the best possible balance between abstraction and observability.

The rest of the paper is organized as follows. Background on membership equational logic and rewriting logic is given in Section 2. The relationship to equational 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, and summarize other language specification case studies in Section 3.2. 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). We end with some concluding remarks in Section 5.

2 Rewriting Logic Semantics

Membership Equational Logic. A membership equational logic (MEL) [33] *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 $A_f \colon A_{k_1} \times \cdots \times A_{k_n} \to A_k$ 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 "$(\forall X)\ \varphi\ \text{ if }\ \bigwedge_i p_i = q_i\ \wedge\ \bigwedge_j w_j : s_j$", 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 [33] 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 "$(\forall x : k)\ x : s_2\ \text{ if }\ x : s_1$". Similarly, an operator declaration $f \colon s_1 \times \cdots \times s_n \longrightarrow s$ corresponds to declaring f at the kind level and giving the membership axiom "$(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s\ \text{ if }\ \bigwedge_{1\leq i\leq n} x_i : s_i$". We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of "$(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t'\ \text{ if }\ \bigwedge_{1\leq i\leq n} x_i : s_i$".

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 [3] 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 [4] *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 $1 \leq i_j \leq n$, which must be fully evaluated (up to the context-sensitive equational reduction strategy specified by μ) in the order specified by the list $i_1 \ldots i_k$ before applying any equations whose lefthand 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 [5] . 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

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

[4] 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.*

[5] Maude has a functional sublanguage whose modules are membership equational theories. Maps μ specifying context-sensitive equational reduction strategies are called *evaluation strategies* [14], and $\mu(f) = i_1 \ldots i_k$ is specified with the `strat` keyword followed by the string $(i_1\ \ldots\ i_k\ 0)$, with 0 indicating evaluation at the top of the function symbol f.

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

$$r : (\forall X)\ t \longrightarrow t' \ \text{ if } \ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w'_l) \qquad (1)$$

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 $\phi(f) = \{i_1, \ldots, i_k\}$, $1 \leq i_1 < \ldots < i_k \leq n$ of *frozen argument positions*[6] 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 $t/\alpha = f(u_1, \ldots, u_i, \ldots, u_n)$, 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 $(\forall X)\ t \longrightarrow t'$,

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

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 $f : k_1 \ldots k_n \longrightarrow k$ in Σ, with $\{1, \ldots, n\} - \phi(f) = \{j_1, \ldots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$,

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

- **Replacement**. For each $\theta : X \longrightarrow T_\Sigma(Y)$ 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,

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

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

$$\frac{\begin{array}{c}(\bigwedge_r (\forall Y)\ p_{j_r} \longrightarrow p'_{j_r})\\ \\ (\bigwedge_i (\forall Y)\ \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y)\ \theta(v_j) : s_j) \wedge (\bigwedge_k (\forall Y)\ \theta(w_k) \longrightarrow \theta(w'_k))\end{array}}{(\forall Y)\ \theta(t) \longrightarrow \theta'(t')}$$

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**. $\dfrac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$

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

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 [60]; 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* [14], meaning that in a rule of the form (1) on page 7, 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 [14] 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* [32,8] 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 [32,8]. The class of models of a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ has an *initial model* $\mathcal{T}_{\mathcal{R}}$ [32,8]. 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 [25], which is the most commonly used form of algebraic 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 knob" 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*

$$Q \longrightarrow Q' \;\; \mathit{if} \;\; P_1 \longrightarrow P_1' \;\wedge\; \ldots \;\wedge\; P_n \longrightarrow P_n'$$

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, but these technical differences present no real difficulty for faithfully expressing SOS within rewriting logic: this is shown in detail in [36].

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) [41,42,43], 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 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 [35] 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) [3], and has been used to give semantics to different concurrent calculi and programming languages (see [3,38] 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* [24] 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 supposed 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 [37] for a complete definition of simple, as well as more details on our language definitional methodology.

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.* context-free[7] mix-fix syntax, we can define the syntax

[7] 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 an mel

of our programming languages in Maude and use its parser generator to parse programs. We show below how to define the syntax of SIMPLE using mix-fix notation. 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, we can also define several common names as constants, so we can omit the quotes to enhance the readability of our programs:

```
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 x y z w : -> Name .
endfm
```

SIMPLE is an *expression language*, meaning that everything parses to an expression. 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 [37] 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

theory or a rewrite theory, its signature part Σ specifies a user-defined context-free grammar for which Maude automatically generates a parser.

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 and personal taste reasons, we do not want to define as a subsort of `Exp`. Consequently, we cannot overload those operators in our SIMPLE language. That is the reason why we added a back-quote to their names in the module above.

Conditionals are indispensable to 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 many consider to be a poor notation):

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

The attribute[8] `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. Since we have a natural subsort structure between the element sorts of these different lists, we can define the corresponding list sorts in a "subtype polymorphism" style. 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
```

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 .
```

[8] 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.

```
  op _,_ : NameList NameList -> NameList [ditto] .
  eq () = nil .
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 gather (e E)] .
```

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 spawn_ lock 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 .  including BOOLEAN-EXP-SYNTAX .
  including IF-SYNTAX .              including ASSIGNMENT-SYNTAX .
  including SEQ-COMP-SYNTAX .        including BLOCK-SYNTAX .
  including LOOPS-SYNTAX .           including PRINT-SYNTAX .
  including FUNCTION-SYNTAX .        including FUNCTION-SET .
  including THREAD-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.

SIMPLE *State Infrastructure.* Any practical programming language needs to invariably consider some notion of *state*. 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). We next informally describe each of the ingredients, without giving formal Maude definitions (the interested reader is referred to [37] 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 [37]), one can formalize the entire state infrastructure as a "nested soup" of such ingredients:

```
fmod SIMPLE-STATE is
... sorts and variables ...
  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
```

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) .
```

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 huge 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]) .
```

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 .
endm  --- end of module SIMPLE-HELPING-OPERATIONS
```

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 unnecessary 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[9]:

```
  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) .
```

The semantic definitions of sequential composition, blocks, loops and printing are explained in detail in [37]; 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) .
```

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 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:

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

```
  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 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) .
```

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 4.1 we show how one can use the exact same definition of SIMPLE to formally *analyze* programs.

3.2 Other Language Case Studies

The SIMPLE language discussed in Section 3.1 illustrates a particular language specification style; but this is just one example within a much broader language specification methodology. A key point worth making is that this methodology *scales up* quite well to real languages with many 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, large fragments of Java 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 [23,21]. A similar Maude specification of the semantics of Scheme at UIUC yields an interpreter with .75 the speed of the standard Scheme interpreter on average for the benchmarks we have tested. In fact, the semantics of large fragments of conventional languages are routinely developed by UIUC graduate students as course projects in a few weeks, including, besides Java, the JVM, and Scheme, languages like (alphabetically), Beta, Haskell, Lisp, LLVM, Pict, Python, Ruby, and Smalltalk. A semantics of a Caml-like language with threads was discussed in detail in [36], and a modular rewriting logic semantics of a subset of CML has been given by Chalub and Braga in [11]. 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 [19]. Other language case studies, all specified in Maude, include BC [6], CCS [58,59,6], CIAO [54], Creol [30], ELOTOS [56], MSR [9,52], PLAN [53,54], and the π-calculus [55].

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*:

(i) a *semi-decision procedure* to find failures of safety properties in a (possibly infinite-state) concurrent program using Maude's `search` command;

(ii) an LTL *model checker* for finite-state programs or program abstractions;

(iii) a *theorem prover* (Maude's ITP [15,16]) 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.

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 in Section 3.1. If one executes that program using the command `rew eval(program)` then most likely one would 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.

SOS'05, ENTCS 156, pp. 27-56. 2006

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: 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. 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. The deadlock above is not the only flaw in the original dining philosophers program. In [37] we consider the slightly modified version of dining philosophers where each philosopher continues to alternatively think and eat forever, and show a starvation problem using Maude's LTL model checker.

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 analysis 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 semantics 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. While this is clearly an interesting research subject, 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 systematically modify the semantics of each language construct as above, thereby easily obtaining a type checker. We have defined several type checkers following this semantic abstraction methodology as part of our programming language courses [46]. Students also developed such type checkers as part of their 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 have investigated, also derived from the semantics of a given 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 axiomatized or 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. Formal definitions of domain specific certifiers built on semantic programming language definitions have been investigated in depth in several places. In [13,12] we discuss such certifiers for the domains of units of measurement and a large fragment of C, in [31] we present a domain-specific certifier for the domain of coordinate frames, and in [47] 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.

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 [35]. 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, taking 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 [17] for a large class of languages would likewise be highly desirable.

Acknowledgement

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 [46], 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, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. Manuscript, June 2005.

[2] D. Basin and G. Denker. Maude versus Haskell: an experimental comparison in security protocol analysis. In K. Futatsugi, editor, *Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications*, volume 36. ENTCS, Elsevier, 2000.

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

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

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

[6] C. Braga and J. Meseguer. Modular rewriting semantics in practice. In *Proc.* WRLA'04, volume 117. ENTCS, Springer, 2004.

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

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

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

[10] 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`.

[11] F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004. `http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics`.

[12] F. Chen and G. Roşu. Certifying measurement unit safety policy. In *Proceedings, International Conference on Automated Software Engineering (ASE'03)*. IEEE, 2003.

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

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

[15] M. Clavel, F. Durán, S. Eker, and 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`.

[16] M. Clavel and M. Palomino. The ITP tool's manual. Universidad Complutense, Madrid, April 2005, `http://maude.sip.ucm.es/itp/`.

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

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

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

[20] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[21] A. Farzan, F. Cheng, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *Proc. CAV'04*, volume 3114 of *Springer LNCS*, 2004.

[22] A. Farzan and J. Meseguer. Partial order reduction for rewriting semantics of programming languages. Technical Report UIUCDCS-R-2005-2598, CS Dept., University of Illinois at Urbana-Champaign, June 2005.

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

[24] M. Felleisen and 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*, pages 193–217. North Holland, 1987.

[25] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.

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

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

[28] P. H. Hartel. LETOS – a lightweight execution tool for operational semantics. *Software: Practice and Experience*, 29:1379–1416, 1999.

[29] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics.* John Willey & Sons, 1990.

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

[31] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 81–90. IEEE, 2001. Coronado Island, California.

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

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

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

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

[36] J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR'04, Cork, Ireland, July 2004*, pages 1–44. Springer LNAI 3097, 2004.

[37] J. Meseguer and G. Roşu. The rewriting logic semantics project. Technical Report UIUCDCS-R-2005-2639, University of Illinois at Urbana-Champaign, 2005.

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

[39] J. Moore, R. Krug, H. Liu, and 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.

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

[41] P. D. Mosses. Foundations of modular SOS. In *Proceedings of MFCS'99, 24th International Symposium on Mathematical Foundations of Computer Science*, pages 70–80. Springer LNCS 1672, 1999.

[42] P. D. Mosses. Pragmatics of modular SOS. In *Proceedings of AMAST'02, 9th Intl. Conf. on Algebraic Methodology and Software Technology*, pages 21–40. Springer LNCS 2422, 2002.

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

[44] M. Pettersson. *Compiling Natural Semantics.* Springer Verlag, LNCS 1549, 1999.

[45] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Previously published as technical report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[46] G. Roşu. Programming language classes. Department of Computer Science, University of Illinois at Urbana-Champaign, http://fsl.cs.uiuc.edu/~grosu/classes/.

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

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

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

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

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

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

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

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

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

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

[57] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.

[58] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In *Proc. FORTE/PSTV 2000*, pages 351–366. IFIP, vol. 183, 2000.

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

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

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