Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools

José Meseguer and Grigore Roşu
University of Illinois at Urbana-Champaign, USA

Abstract. Formal semantic definitions of concurrent languages, when specified in a well-suited semantic framework and supported by generic and efficient formal tools, can be the basis of powerful software analysis tools. Such tools can be obtained *for free* from the semantic definitions; in our experience in just the few weeks required to define a language's semantics even for large languages like Java. By combining, yet distinguishing, both equations and rules, rewriting logic semantic definitions unify both the semantic equations of equational semantics (in their higher-order denotational version or their first-order algebraic counterpart) and the semantic rules of SOS. Several limitations of both SOS and equational semantics are thus overcome within this unified framework. By using a high-performance implementation of rewriting logic such as Maude, a language's formal specification can be automatically transformed into an efficient interpreter. Furthermore, by using Maude's breadth first search command, we also obtain for free a semi-decision procedure for finding failures of safety properties; and by using Maude's LTL model checker, we obtain, also for free, a decision procedure for LTL properties of finite-state programs. These possibilities, and the competitive performance of the analysis tools thus obtained, are illustrated by means of a concurrent Caml-like language; similar experience with Java (source and JVM) programs is also summarized.

1 Introduction

Without a precise *mathematical semantics* compiler writers will often produce incompatible language implementations; and it will be meaningless to even attempt to formally verify a program. Formal semantics is not only a necessary *prerequisite* to any meaningful talk of software correctness, but, as we try to show in this paper, it can be a key technology to develop powerful software analysis tools. However, for this to happen in practice we need to have:

- a well-suited semantic framework, and
- a high performance implementation of such a framework.

We argue that rewriting logic is indeed a well-suited and flexible framework to give formal semantics to programming languages, including concurrent ones. In fact we show that it unifies two well-known frameworks, namely equational semantics and structural operational semantics, combining the advantages of both and overcoming several of their respective limitations.

High performance is crucial to scale up both the execution and the formal analysis. In this regard, the existence of the Maude 2.0 system [18] implementing

rewriting logic and supporting efficient execution as well as breadth-first search and LTL model checking, allows us to automatically turn a language's rewriting logic semantic definition into a quite sophisticated software analysis tool for that language *for free*. In particular, we can efficiently interpret programs in that language, and we can formally analyze programs, including concurrent ones, to find safety violations and to verify temporal logic properties by model checking.

The fact that rewriting logic specifications provide in practice an easy way to develop executable formal definitions of languages, which can then be subjected to different tool-supported formal analyses, is by now well established [82, 7, 83, 77, 73, 44, 79, 15, 64, 80, 26, 25, 37]. However, ascertaining that this approach can scale up to large conventional languages such as Java and the JVM [26, 25], and that the generic formal analysis methods associated to semantic definitions can compete in performance with special-purpose analysis tools developed for individual languages, is a more recent development that we have been investigating with our students and for which we give evidence in this paper.

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 (denotational semantics) and first-order (algebraic semantics) versions have been shown to be useful formalisms. There is a vast literature in these two areas that we do not attempt to survey. However, we can mention some early denotational semantics papers such as [74, 66] and the survey [55]. Similarly, we can mention [88, 30, 11] for early algebraic semantics papers, and [29] for a recent textbook.

We use the more neutral term *equational semantics* to emphasize the fact that denotational and algebraic semantics have many common features and can both be viewed as instances of a common equational framework. In fact, there isn't a rigid boundary between both approaches, as illustrated, for example, by the conversion of higher-order semantic equations into first-order ones by means of explicit substitution calculi or combinators, the common use of initiality in both initial algebras and in solutions of domain equations, and a continuous version of algebraic semantics based on continuous algebras.

Strong points of equational semantics include:

- it has a *model-theoretic*, denotational semantics given by *domains* in the higher-order case, and by *initial algebras* in the first-order case;
- it has also a *proof-theoretic*, operational semantics given by *equational reduction* with the semantic equations;
- 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.);
- there is good higher-order and first-order theorem proving support.

However, equational semantics has the following drawbacks:

- it is well suited for *deterministic* languages such as conventional sequential languages or purely functional languages, but is quite poorly suited to define the semantics of *concurrent languages*, unless the concurrency is that of a purely deterministic computation;
- 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 within an equational framework;
- 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 componenta. The "locus classicus" is Plotkin's Aarhus lectures [61]; there is again a vast literature on the topic that we do not attempt to survey; for a good textbook introduction see [34].

Strong points of SOS include:

- it is an abstract and general formalism, yet quite intuitive, allowing detailed *step-by-step* modeling of program execution;
- has a simple *proof-theoretic* semantics using semantic rules as inference rules;
- is fairly well suited to model *concurrent languages*, and can also deal well with the detailed execution of deterministic languages;
- allows *mathematical reasoning and proof*, by reasoning inductively or coinductively about the inference steps.

However, SOS has the following drawbacks:

- although specific proposals have been made for *categorical models* for certain SOS formats, such as, for example, Turi's functorial SOS [78] and Gadducci and Montanari's tile models [28], it seems however fair to say that, so far, SOS has not commonly agreed upon model-theoretic semantics.
- 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 5.2).
- although some tools have been built to execute SOS definitions (see for example [20]) tool support is considerably less developed than for equational semantics.
- standard SOS definitions are notoriously *unmodular*, unless one adopts Mosses' MSOS framework (see Section 5.3).

[1] Two good examples of indirectly modeling concurrency within a purely functional framework are the ACL2 semantics of the JVM using a scheduler [52], and the use of lazy data structures in Haskell to analyze cryptographic protocols [2].

1.2 Rewriting Logic Unifies SOS and Equational Semantics

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 seems 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.

This paper addresses three fundamental questions:

1. can the semantic frameworks of SOS and equational semantics be unified in a mathematically rigorous way?
2. can the advantages of each formalisms be preserved and can their respective drawbacks be overcome in a suitable unification?
3. is it possible to efficiently execute and analyze programs using semantic language definitions in such a unified framework with suitable formal tools?

We answer each of the above questions in the affirmative by proposing rewriting logic [40, 13] as such a unifying semantic framework. Roughly speaking,[2] a rewrite theory is a triple (Σ, E, R), with (Σ, E) an equational theory with signature of operations and sorts Σ and set of (possibly conditional) equations E, and with R a set of (possibly conditional) rewrite rules. Therefore, rewriting logic introduces a *key distinction* between semantic *equations* E, and semantic *rules* R. Computationally, this is a distinction between *deterministic* computations, and *concurrent* and possibly nondeterministic ones. That is, if (Σ, E, R) axiomatizes the semantics of a programming language $\mathcal{L}$, then the deterministic computations in $\mathcal{L}$ will be axiomatized by the semantic equations E, whereas the concurrent computations will be axiomatized by the rewrite rules R. The semantic unification of SOS and equational semantics is then achieved very naturally, since, roughly speaking, we can obtain SOS and equational semantics as, respectively, the special cases in which $E = \varnothing$ and we have only semantic rules[3], and $R = \varnothing$ and we have only semantic equations, respectively.

This unification makes possible something not available in either formalism, namely mixing semantic equations and semantic rules, using each kind of axiom for the purposes for which it is best suited: equations for deterministic computations, and rules for concurrent ones. This distinction between equations and rules is of more than academic interest. The point is that, since rewriting with rules R takes place *modulo* the equations E [40], many states are abstracted together by the equations E, and only the rules R contribute to the size of the system's state space, which can be drastically smaller than if all axioms had been given as rules, a fact of crucial importance for formal analyses of concurrent programs based on search and model checking.

---

[2] We postpone the issue of "frozen" arguments, which is treated in Section 2.2.

[3] The case of structural axioms is a separate issue that we postpone until Section 2; also rewrite rules and SOS rules, though closely related, do not correspond identically to each other, as explained in Section 5.2.

This brings us to efficient tool support for both execution and formal analysis. Rewriting logic has several high-performance implementations [5, 27, 18], of which the most comprehensive so far, in expressiveness and in range of features, is probably the Maude system [18]. Maude can both efficiently execute a rewriting logic axiomatization of a programming language $\mathcal{L}$, thus providing an interpreter for $\mathcal{L}$, and also perform breadth-first search to find safety violations in a concurrent program, and model checking of linear time temporal logic (LTL) properties for such programs when the set of reachable states is finite. We illustrate these execution and analysis capabilities in Sections 3–4.

The rest of the paper is organized as follows. Basic concepts on rewriting logic and membership equational logic are gathered in Section 2. We then illustrate our language specification methodology by means of a nontrivial example – a substantial Caml-like language including functions, assignments, loops, exceptions, and threads – and briefly discuss another case study on Java semantics in Section 3. The formal analysis of concurrent programs is illustrated for our example language in Section 4. We revisit SOS and equational semantics and discuss the advantages of their unification within rewriting logic in Section 5. The paper gives some concluding remarks in Section 6.

2 Rewriting Logic Semantics

We explain here the basic concepts of rewriting logic, and how it can be used to define the semantics of a programming language. Since each rewrite theory has an underlying equational theory, different variants of equational logic give rise to corresponding variants of rewriting logic. The more expressive the underlying equational sublanguage, the more expressive will the resulting rewrite theories be. For this reason, we include below a brief summary of membership equational logic (MEL) [43], an expressive Horn logic with both equations $t = t'$ and membership predicates $t : s$ which generalizes order-sorted equational logic and supports sorts, subsorts, partiality, and sorts defined by equational axioms. Maude 2.0 [18] supports all the logical features of MEL and its rewriting logic super-logic with a syntax almost identical to the mathematical notation.

2.1 Membership Equational Logic

A membership equational logic (MEL) [43] *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 $(\forall X)\ \varphi \text{ if } \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$, 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 [43] for the detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, and initial and free algebras. In particular, given an 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 $\bigwedge_{1 \leq i \leq n} x_i : s_i$. We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of $(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t'$ if $\bigwedge_{1 \leq i \leq n} x_i : s_i$.

2.2 Rewrite Theories

A *rewriting logic specification or theory* is a tuple $\mathcal{R} = (\Sigma, E, \phi, R)$, with:

- (Σ, E) a membership equational theory
- $\phi : \Sigma \longrightarrow \mathbb{N}$ 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* under which it is forbidden to perform any rewrites; and
- R a set of *labeled conditional rewrite rules* of the general form

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

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 $[s_j]$. Intuitively, $\mathcal{R}$ specifies a *concurrent system*, whose states are elements of the initial algebra $T_{\Sigma/E}$ specified by (Σ, E), and whose *concurrent transitions* are specified by the rules R, subject to the frozenness imposed by ϕ.

We can illustrate both a simple rewrite theory and the usefulness of frozen arguments by means of the following Maude module for nondeterministic choice:

```
mod CHOICE is protecting NAT .
  sorts Elt MSet .  subsorts Elt < MSet .
  ops a b c d e f g : -> Elt .
  op __ : MSet MSet -> MSet [assoc comm] .
  op card : MSet -> Nat [frozen] .
  eq card(X:Elt) = 1 .
  eq card(X:Elt M:MSet) = 1 + card(M:MSet) .
  rl [choice] : X:MSet Y:MSet => Y:MSet .
endm
```

In a Maude module,[4] introduced with the keyword `mod` followed by its name, and ended with the keyword `endm`, kinds are not declared explicitly; instead, each connected component of sorts in the subsort inclusion ordering implicitly determines a kind, which is viewed as the equivalence class of its corresponding sorts. Here, since the only two sorts declared, namely `Elt` and `MSet`, are related by a subsort inclusion[5] we have implicitly declared a new kind, which we can

[4] A Maude module specifies a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$; however, when $R = \varnothing$, then $\mathcal{R}$ becomes a membership equational theory. Maude has an *equational sublanguage* in which a membership equational theory (Σ, E) can be specified as a *functional module* with beginning and ending keywords `fmod` and `endfm`.

[5] A subsort inclusion is shorthand notation for a corresponding membership axiom.

refer to by enclosing either of the sorts in square brackets, that is, by either [Elt] or [MSet]. There are however two more kinds, namely the kind [Nat] determined by the sort Nat of natural numbers and its subsorts in the imported NAT module, and the kind [Bool] associated to the Booleans, which by default are implicitly imported by any Maude module.

The operators in Σ are declared with op (ops if several operators are declared simultaneously). Here we have just three such declarations: (i) the constants a through g of sort Elt, (ii) a multiset union operator declared with infix[6] empty syntax (juxtaposition), and (iii) a multiset cardinality function declared with prefix notation card. The set E contains those equations and memberships of the imported modules, the two equations defining the cardinality function, and the equations of associativity and commutativity for the multiset union operator, which are not spelled out as the other equations, but are instead specified with the assoc and comm keywords. Furthermore, as pointed out in Section 2.1, the subsort inclusion declaration and the operator declarations at the level of sorts are in fact conditional membership axioms in disguise. The only rule in the set R is the [choice] rule, which arbitrarily chooses a nonempty sub-multiset of the given multiset. Maude then uses the assoc and comm declarations to apply the other equations and the [choice] rule in a built-in way *modulo* the associativity and commutativity of multiset union, that is, parentheses are not needed, and the order of the elements in the multiset is immaterial. It is then intuitively clear that if we begin with a multiset such as a a b b b c and repeatedly rewrite it in all possible ways using the [choice] rule, the terminating (deadlock) states will be the singleton multisets a, b, and c.

The multiset union operator has no special declaration, meaning that none of its two arguments are frozen, but the cardinality function is declared with the frozen attribute, meaning that all its arguments (in this case the single argument of sort MSet) are frozen, that is, $\phi(\texttt{card}) = \{1\}$. This declaration captures the intuition that it does not make much sense to rewrite below the cardinality function card, because then the multiset whose cardinality we wish to determine would become a *moving target*. If card had not been declared frozen, then the rewrites a b c $\longrightarrow$ b c $\longrightarrow$ c would induce rewrites $3 \longrightarrow 2 \longrightarrow 1$, which seems bizarre. The point is that we think of the kind [MSet] as the *state kind* in this example, whereas [Nat] is the *data kind*. By declaring card's single argument as frozen, we restrict rewrites to the state kind, where they belong.

2.3 Rewriting Logic Deduction

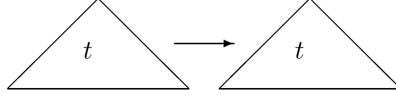
Given $\mathcal{R} = (\Sigma, E, \phi, R)$, the sentences that it proves are universally quantified rewrites of the form, $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_{\Sigma,E}(X)_k$, for some kind k, which are obtained by finite application of the following *rules of deduction*:

[6] In general, prefix, infix, postfix, and general "mixfix" user-definable syntax is supported. In all cases except for prefix operators, each argument position is declared with an underbar symbol; for example, the usual infix notation for addition would be declared _+_, but here, since we use juxtaposition, no symbol is given between the two underbars __ of the multiset union operator.

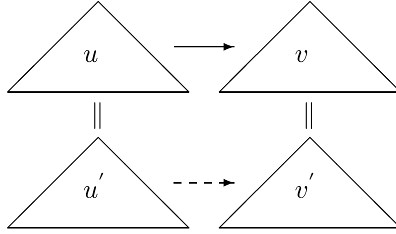
- **Reflexivity**. For each $t \in T_\Sigma(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$
- **Equality**. $\dfrac{(\forall X)\ u \longrightarrow v \quad E \vdash (\forall X)u = u' \quad E \vdash (\forall X)v = v'}{(\forall X)\ u' \longrightarrow v'}$
- **Congruence**. For each $f : k_1 \dots k_n \longrightarrow k$ in Σ, with $\{1, \dots, n\} - \phi(f) = \{j_1, \dots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$,
$$\frac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad \dots \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, \dots, t_{j_1}, \dots, t_{j_m}, \dots, t_n) \longrightarrow f(t_1, \dots, t'_{j_1}, \dots, t'_{j_m}, \dots, t_n)}$$
- **Replacement**. For each $\theta : X \longrightarrow T_\Sigma(Y)$ with, say, $X = \{x_1, \dots, x_n\}$, 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 $Z = \{x_{j_1}, \dots, x_{j_m}\}$ the set of unfrozen variables in t and t', then,
$$\frac{(\bigwedge_r (\forall Y)\ p_{j_r} \longrightarrow p'_{j_r}) \quad (\bigwedge_i (\forall Y)\ \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y)\ \theta(v_j) : s_j) \wedge (\bigwedge_k (\forall Y)\ \theta(w_k) \longrightarrow \theta(w'_k))}{(\forall Y)\ \theta(t) \longrightarrow \theta'(t')}$$
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** $\dfrac{(\forall X)\ t_1 \longrightarrow t_2 \quad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$

We can visualize the above inference rules as follows:

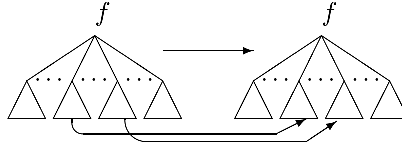
Reflexivity



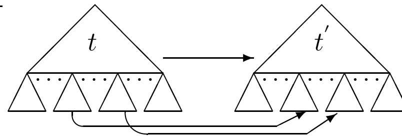
Equality



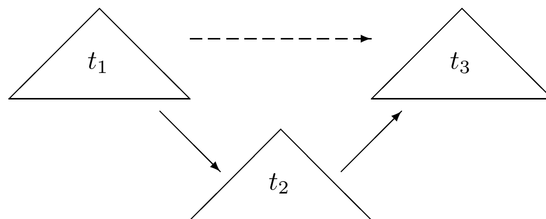
Congruence



Replacement



<u>*Transitivity*</u>



Intuitively, we should think of the above 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 a rewrite theory to be *executable*, so that the above inference rules can be efficiently tool supported, some additional requirements should be met. First, E should decompose as a union $E = E_0 \cup A$, with A a set of equational axioms such as associativity, commutativity, identity, for which an effective matching algorithm modulo A exists, and E_0 a set of (ground) confluent and terminating[7] for each term t by applying the equations in E_0 modulo A to t until termination. Second, the rules R should be *coherent* with E_0 modulo A [86]; intuitively, this means that, to get the effect of rewriting in equivalence classes modulo E, we can always first simplify a term with the equations to its canonical form, 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 ($\flat$), 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).

[7] The termination condition may be dropped for programming language specifications in which some equationally defined language constructs may not terminate. Even the confluence modulo A may be relaxed, by restricting it to terms in some "observable kinds" of interest. The point is that there may be some "unobservable" kinds for which several different but semantically equivalent terms can be derived by equational simplification: all we need in practice is that the operations are confluent for terms in an observable kind, such as that of values, so that a unique canonical form is then reached for them if it exists.

2.4 Rewriting Logic's Model Theory and Temporal Logic

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, its *$\mathcal{R}$-reachability relation* $\rightarrow_{\mathcal{R}}$ (also called *$\mathcal{R}$-rewriting relation*, or *$\mathcal{R}$-provability relation*) is defined proof-theoretically, for each kind k in Σ and each $[t], [t'] \in T_{\Sigma/E,k}$, by the equivalence,

$$[t] \rightarrow_{\mathcal{R}} [t'] \quad \Leftrightarrow \quad \mathcal{R} \vdash (\forall\varnothing)\, t \longrightarrow t',$$

which by the **Equality** rule is independent of the choice of t, t'. Model-theoretically, *$\mathcal{R}$-reachability* can be defined as the family of relations, indexed by the kinds k in Σ, interpreting the sorts $Arrow_k$ in the initial model of a membership equational theory $Reach(\mathcal{R})$ axiomatizing the *reachability models* of the rewrite theory $\mathcal{R}$ [13]. The initial reachability model is then the initial algebra $T_{Reach(\mathcal{R})}$. In particular, the *one-step $\mathcal{R}$-rewrite relations* for each kind k are the extensions in $T_{Reach(\mathcal{R})}$ of subsorts $Arrow^1_k < Arrow_k$. We denote such a relation on E-equivalence classes of terms with the notation $[t] \rightarrow^1_{\mathcal{R},k} [t']$. Thus, a rewrite theory $\mathcal{R}$ specifies for each kind k a transition system characterized by $\rightarrow^1_{\mathcal{R},k}$, which can be made total by adding idle transitions for deadlock states, denoted $(\rightarrow^1_{\mathcal{R},k})^\bullet$. This is almost a Kripke structure: we still need to specify the *state predicates* in a set of predicates Π. This can be done equationally, by choosing a kind k as the kind of states, and giving equations defining when each predicate $p \in \Pi$ holds for a state $[t]$ of sort k, thus getting a labeling function L_Π.

This way, we can associate to a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ with a designated kind k of states and state predicates Π, the Kripke structure $(T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R},k})^\bullet, L_\Pi)$. We can then define the semantics of any temporal logic formula over predicates Π in the usual way [16], for any desired temporal logic such as LTL, CTL*, the modal μ-calculus, and so on (see [44]). Furthermore, if the states reachable from an initial state form a finite set, then we can model check such formulas. Maude provides an explicit state LTL model checking for executable rewrite theories with a performance comparable to that of SPIN [23].

Reachability models for a rewrite theory $\mathcal{R}$ are a special case of more general *true concurrency models*, in which different concurrent computations from one state to another correspond to *equivalence classes of proofs* in rewriting logic. That is, concurrent computations are placed in bijective correspondence with proofs in a Curry-Howard like equivalence. The paper [13] shows that initial models exist for both reachability models and true concurrency models of a rewrite theory $\mathcal{R}$, and that both kinds of models make the rules of inference of rewriting logic sound and complete. We denote by $T_{Reach(\mathcal{R})}$, resp. $\mathcal{T}_{\mathcal{R}}$, the initial reachability, resp. true-concurrency, model of a rewrite theory $\mathcal{R}$.

2.5 Specifying Concurrency Models and Programming Languages

Because rewriting logic is *neutral* about concurrency constructs, it is a general *semantic framework* for concurrency that can express many concurrency models such as: equational programming, which is the special case of rewrite theories whose set of rules is empty and whose equations are Church-Rosser, possibly modulo some axioms A; lambda calculi and combinatory reduction systems

[40, 38, 71, 68]; labeled transition systems [40]; grammars and string-rewriting systems [40]; Petri nets, including place/transition nets, contextual nets, algebraic nets, colored nets, and timed Petri nets [40, 42, 69, 72, 59, 67]; Gamma and the Chemical Abstract Machine [40]; CCS and LOTOS [47, 39, 14, 22, 82, 81, 83, 79]; the π calculus [84, 68, 77]; concurrent objects and actors [40, 41, 75, 76]; the UNITY language [40]; concurrent graph rewriting [42]; dataflow [42]; neural networks [42]; real-time systems, including timed automata, timed transition systems, hybrid automata, and timed Petri nets [59, 58]; and the tile logic [28] model of synchronized concurrent computation [48, 12].

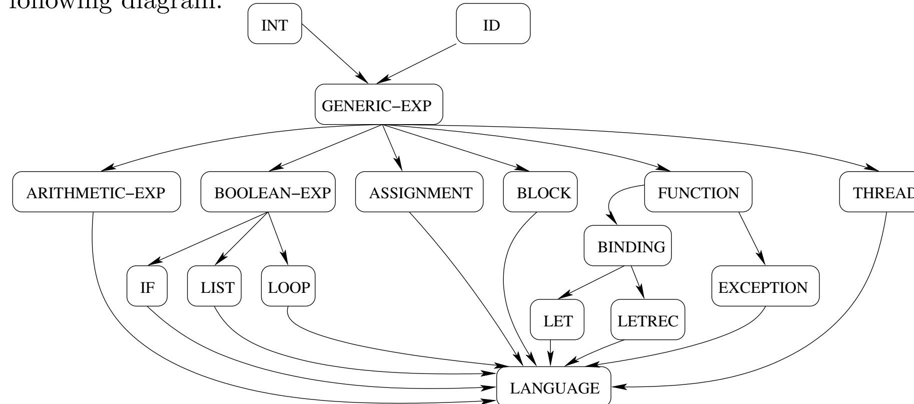
Since the above are typically executable, rewriting logic is a flexible *operational* semantic framework to specify such models. What is not immediately apparent is that it is also a flexible *mathematical* semantic framework for concurrency models. Well-known models of concurrency are isomorphic to the initial model $\mathcal{T}_\mathcal{R}$ of the rewrite theory $\mathcal{R}$ axiomatizing that particular model, or at least closely related to such an initial model: [38] shows that for rewrite theories $\mathcal{R} = (\Sigma, \varnothing, \phi, R)$ with the rules R left-linear, $\mathcal{T}_\mathcal{R}$ is isomorphic to a model based on residuals and permutation equivalence proposed by Boudol [6], and also that for $\mathcal{R}$ a rewrite theory of an orthogonal combinatory reduction system, including the λ-calculus, a quotient of $\mathcal{T}_\mathcal{R}$ is isomorphic to a well-known model of parallel reductions; [72] shows that for $\mathcal{R}$ a rewrite theory of a place/transition net, $\mathcal{T}_\mathcal{R}$ is isomorphic to the Best-Devillers net process model [4] and then generalizes this isomorphism to one between $\mathcal{T}_\mathcal{R}$ and a Best-Devillers-like model for the rewrite theory of an algebraic net; [14, 22] show that for $\mathcal{R}$ axiomatizing CCS, a truly concurrent semantics causal model based on proved transition systems is isomorphic to a quotient of $\mathcal{T}_\mathcal{R}$; [49] shows that for $\mathcal{R}$ axiomatizing a concurrent object-oriented system satisfying reasonable requirements, a subcategory of $\mathcal{T}_\mathcal{R}$ is isomorphic to a partial order of events model which, for asynchronous object systems corresponding to actors, coincides with the finitary part of the Baker-Hewitt partial order of events model [1].

All the above remarks apply also to the specification of programming languages, which often implement specific concurrency models. In particular, both an operational semantics and a denotational semantics are provided for a language when it is specified as a rewrite theory. How is this generally done? We can define the semantics of a concurrent programming language, say $\mathcal{L}$, by specifying a rewrite theory, say $\mathcal{R}_\mathcal{L} = (\Sigma_\mathcal{L}, E_\mathcal{L}, \phi_\mathcal{L}, R_\mathcal{L})$, where $\Sigma_\mathcal{L}$ is $\mathcal{L}$'s *syntax* and the auxiliary operators (store, environment, etc.), $E_\mathcal{L}$ specifies the semantics of all the *deterministic features* of $\mathcal{L}$ and of the auxiliary semantic operations, the frozenness information $\phi_\mathcal{L}$ specifies what arguments can be rewritten with rules for each operator, and the rewrite rules $R_\mathcal{L}$ specify the semantics of all the *concurrent features* of $\mathcal{L}$. Section 3 does exactly this.

3 Specifying Deterministic and Concurrent Features

In this section we illustrate the rewriting logic semantics techniques advocated in this paper on a nontrivial Caml-like programming language. We show how several important programming language features, such as arithmetic and boolean

expressions, conditional statements, high-order functions, lists, `let` bindings, recursion with `let rec`, side effects via variable assignments, blocks and loops, exceptions, and concurrency via threads and synchronization, can be succinctly, modularly and efficiently defined in rewriting logic. What we present in this section should be regarded as one possible way to define this language, a way which is by no means unique or optimal. The various features are shown in the following diagram:



`INT` is a Maude builtin module defining arbitrary precision integers; `ID` defines identifiers as well as comma-separated lists of identifiers; `GENERIC-EXP` defines a special sort for expressions as well as comma-separated lists of such expressions; `ARITHMETIC-EXP` adds arithmetic operators, such as addition, multiplication, etc., and `BOOLEAN-EXP` adds boolean expressions; the latter are further used to define conditionals, loops and lists (lists contain an empty list check); `BINDINGS` defines bindings as special lists of pairs "identifier = expression", which are further needed to define both `LET` and `LETREC`; `ASSIGNMENT` defines variable assignments; `BLOCK` defines blocks enclosed with curly brackets "{" and "}" containing sequences of expressions separated by semicolon ";" (blocks are used for their side effects); `FUNCTION` defines high order functions, in a Caml style; `EXCEPTION` defines exceptions using `try ... catch ...` and `throw ...` keywords, where the "catch" part is supposed to evaluate to a function of one argument, to which a value can be "thrown" from the "try" part; `THREAD` defines new threads which can be created and destroyed dynamically; finally, `LANGUAGE` creates the desired programming language by putting all the features together. Each of the above has a syntactic and a semantic part, each specified as a Maude module. The entire Maude specification has less than 400 lines of code.

3.1 Defining the Syntax and Desugaring

Here we show how to define the syntax of a programming language in Maude, together with several simple desugaring translations that will later simplify the semantic definitions, such as translations of "for" loops into "while" loops. We first define identifiers, which add to Maude's builtin quoted identifiers (`QID`) several common (unquoted) one-character identifiers, together with comma-separated lists of identifiers that will be needed later:

```
fmod ID is protecting QID .
  sorts Id IdList .  subsorts Qid < Id < IdList .
  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 : -> Id .
  op nil : -> IdList .
  op _,_ : IdList IdList -> IdList [assoc id: nil prec 1] .
endfm
```

The attribute "`prec 1`" assigns a precedence to the comma operator, to avoid using unnecessary parentheses: the lower the precedence of an operator the tighter the binding. We next define generic expressions, including for now integers and white-space-separated sequences of "names", where a name is either an identifier or the special symbol "`()`". Sequences of names and comma-separated lists of expressions will be needed later to define lists and bindings and function declarations, respectively. The attribute "`gather(E e)`" states that the name sequencing operator is left associative and the "`ditto`" attribute states that the current operation inherits all the attributes of an operation with the same name and kind arrity previously defined (in our case the comma operator in `ID`):

```
fmod GENERIC-EXP-SYNTAX is protecting ID . protecting INT .
  sorts Unit Name NameSeq Exp ExpList .
  subsorts Unit Id < Name < NameSeq < Exp < ExpList .
  subsort Int < Exp .  subsort IdList < ExpList .
  op `(`) : -> Unit .
  op __ : NameSeq NameSeq -> NameSeq [gather(E e) prec 1] .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm
```

The rest of the syntax adds new expressions to the language modularly. The next four modules add arithmetic, boolean, conditional and list expressions. Note that in the `LIST` module, the list constructor takes a comma-separated list of expressions and returns an expression:

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

fmod BOOLEAN-EXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  ops _==_ _<=_ _>=_ _<_ _>_ _and_ _or_ : Exp Exp -> Exp .
  op not_ : Exp -> Exp .
endfm

fmod IF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list : ExpList -> Exp .
  ops car cdr null? : Exp -> Exp .
  op cons : Exp Exp -> Exp .
endfm
```

We next define functions. Like in Caml, we want to define functions using a syntax like "`fun x y z -> x + y * z`". However, "`fun x y z -> ...`" is syntactic sugar for "`fun x -> fun y -> fun z -> ...`", so to keep the semantics simple later we prefer to consider this uncurrying transformation as part of the syntax. Function application simply extends the name sequencing operator:

```
fmod FUNCTION-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op fun_->_ : NameSeq Exp -> Exp .
  op __ : Exp Exp -> Exp [ditto] .
  var Zs : NameSeq .  var Z : Name .  var E : Exp .
  eq fun Zs Z -> E = fun Zs -> fun Z -> E .
endfm
```

Bindings of names to values are crucial in any functional programming language. Like in Caml, in our language bindings are "`and`"-separated pairs of equalities. However, note that "`f x y z = ...`" is just syntactic sugar for "`f = fun x y z -> ...`". Since the semantics of bindings will involve allocation of new memory locations for the bound identifiers, it will be very helpful to know upfront the number and the list of identifiers. Two equations take care of this:

```
fmod BINDING-SYNTAX is extending FUNCTION-SYNTAX .
  sorts Binding Bindings .  subsort Binding < Bindings .
  op _and_ : Bindings Bindings -> Bindings [assoc prec 100] .
  op `(_,_,_`) : Nat IdList ExpList -> Bindings .
  op _=_ : NameSeq Exp -> Binding .
  var Zs : NameSeq .  var Z : Name .  var X : Id .  var E : Exp .
  vars N N' : Nat .  vars Xl Xl' : IdList .  vars El El' : ExpList .
  eq (Zs Z = E) = (Zs = fun Z -> E) .
  eq (X = E) = (1, X, E) .
  eq (N, Xl, El) and (N', Xl', El') = (N + N', (Xl,Xl'), (El,El')) .
endfm
```

We can now define the two major binding language constructors, namely "`let`" and "`let rec`", the later typically being used to define recursive functions:

```
fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : Bindings Exp -> Exp .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op let rec_in_ : Bindings Exp -> Exp .
endfm
```

We next add several imperative features, such as variable assignment, blocks and loops. The variable assignment assumes the identifier already allocated at some location, and just changes the value stored at that location. Both "`for`" and "`while`" loops are allowed, but the former ones are immediately desugared:

```
fmod ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _:=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpBlock .  subsort Exp < ExpBlock .
  op _;_ : ExpBlock ExpBlock -> ExpBlock [assoc prec 100] .
  op {_} : ExpBlock -> Exp .
endfm

fmod LOOP-SYNTAX is extending BLOCK-SYNTAX .
  op while__ : Exp Exp -> Exp .
  op for(_;_;_)_ : Exp Exp Exp Exp -> Exp .
  vars Start Cond Step Body : Exp .
  eq for(Start ; Cond ; Step) Body = {Start ; while Cond {Body ; Step}} .
endfm
```

We next add syntax for two important features, exceptions and threads:

```
fmod EXCEPTION-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op try_catch_ : Exp Exp -> Exp .
  op throw_ : Exp -> Exp .
endfm

fmod THREAD-SYNTAX is extending GENERIC-EXP-SYNTAX .
  ops spawn_ lock acquire_ release_ : Exp -> Exp .
endfm
```

We can now put all the syntax together, noticing that the syntax modules of most of the features above are independent from each other, so one can easily reuse them to build other languages, using Maude 2.01's renaming facility to adapt each module to the concrete syntax of the chosen language:

```
fmod LANGUAGE-SYNTAX is  extending ARITHMETIC-EXP-SYNTAX . extending BOOLEAN-EXP-SYNTAX .
  extending IF-SYNTAX .  extending FUNCTION-SYNTAX .       extending LIST-SYNTAX .
  extending LET-SYNTAX . extending LETREC-SYNTAX .         extending ASSIGNMENT-SYNTAX .
  extending LOOP-SYNTAX . extending EXCEPTION-SYNTAX .     extending THREAD-SYNTAX .
endfm
```

One can now parse programs using Maude's "`parse`" command. For example, the following program recursively calculating the product of elements in a list, will correctly parse as "`Exp`". Note that this program uses an exception to immediately return 0 whenever a 0 is encountered in the input list.

```
parse
  let p l = try let rec a l = if null?(l) then 1
                              else if car(l) == 0 then throw 0
                                   else car(l) * (a cdr(l))
                in a l catch fun x -> x
  in p list(1,2,3,4,5,6,7,8,9,0,10,11,12,13,14,15,16,17,18,19,20)
 .
```

3.2 Defining the State Infrastructure

Before defining the semantics of a programming language, one needs to define the notion of programming language state. There are different possibilities to design the state needed to give semantics to a language, depending on its complexity and one's taste. However, any language worth its salt supports identifiers that are bound to values; since our language has side effects (due to variable assignments), we need to split the mapping of identifiers to values into a map of identifiers to locations, called an environment, and a map of locations to values, called a store.

Let us first define *locations*. A location is essentially an integer; to keep it distinct from other integers, we wrap it with the constructor "`loc`". An auxiliary operation creating a given number of locations starting with a given one will be very useful when defining bindings and functions, so we provide it here.

```
fmod LOCATION is protecting INT .
  sorts Location LocationList .  subsort Location < LocationList .
  op loc : Nat -> Location .
  op nil : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: nil] .
  op locs : Nat Nat -> LocationList .
  eq locs(N:Nat,0) = nil .
  eq locs(N:Nat,#:Nat) = loc(N:Nat), locs(N:Nat + 1, #:Nat - 1) .
endfm
```

An elegant and efficient way to define a mapping in Maude is as a set of pairs formed with an associative (A) and commutative (C) union operator `_||_` with identity (I) `empty`. Then *environments* can be defined as below. Note the use of ACI matching to evaluate or update an environment at an identifier, so that one does not need to traverse the entire set in order to find the desired element:

```
fmod ENVIRONMENT is protecting ID .  protecting LOCATION .
  sort Env .
  op empty : -> Env .
  op [_,_] : Id Location -> Env .
  op _||_ : Env Env -> Env [assoc comm id: empty] .
```

```
  op _[_<-_] : Env IdList LocationList -> Env .
  vars Env : Env . vars L L' : Location . var Xl : IdList . var Ll : LocationList .
  eq Env[nil <- nil] = Env .
  eq ([X:Id,L] || Env)[X:Id,Xl <- L',Ll] = ([X:Id,L'] || Env)[Xl <- Ll] .
  eq Env[X:Id,Xl <- L,Ll] = (Env || [X:Id,L])[Xl <- Ll] [owise] .
endfm
```

Values and *stores* can be defined in a similar way. Since we want our modules to be as independent as possible, to be reused for defining other languages, we prefer to *not* state at this moment the particular values that our language handles, such as integers, booleans, functions (i.e., closures), etc.. Instead, we define the values when they are first needed within the semantics. However, since lists of values are frequently used for various reasons, we believe that many languages need them so we introduce them as part of the `VALUE` module:

```
fmod VALUE is sorts Value ValueList .  subsort Value < ValueList .
  op noValue : -> Value .
  op nil : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: nil] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .  protecting VALUE .
  sort Store .
  op empty : -> Store .
  op [_,_] : Location Value -> Store .
  op _||_ : Store Store -> Store [assoc comm id: empty] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var M : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq M[nil <- nil] = M .
  eq ([L,V] || M)[L,Ll <- V',Vl] = ([L,V'] || M)[Ll <- Vl] .
  eq M[L,Ll <- V',Vl] = (M || [L,V'])[Ll <- Vl] [owise] .
endfm
```

Since our language has complex control-context constructs, such as exceptions and threads, we follow a *continuation passing style (CPS)* definitional methodology (see [62] for a discussion on several independent discoveries of continuations). The use of continuations seems to be novel in the context of semantic language definitions based on algebraic specification techniques. We have found continuations to be very useful in several of our programming language definitions, not only because they allow us to easily and naturally handle complex control-related constructs, but especially because they lead to an increased efficiency in simulations and formal analysis, sometimes more than an order of magnitude faster than using other techniques. Like for values, at this moment we prefer to avoid defining any particular continuation items; we only define the "`stop`" continuation, which will stop the computation, together with the essential operator stacking continuation items on top of an existing continuation:

```
fmod CONTINUATION is sorts Continuation ContinuationItem .
  op stop : -> Continuation .
  op _->_ : ContinuationItem Continuation -> Continuation .
endfm
```

We are now ready to put all the state infrastructure together and to define the *state* of a program in our language. A key decision in our definitional methodology is to consider states as sets of *state attributes*, which can be further nested at any degree required by the particular language definition. This way, the semantic equations and rules will be local, and will only have to mention those

state attributes *that are needed* to define the semantics of a specific feature, thus increasing the clarity, modularity and efficiency of language definitions. The following specifies several state attributes, which are so common in modern languages that we define them as part of the generic state module. Other attributes will be defined later, as needed by specific language features.

```
fmod STATE is sorts StateAttribute LState . subsort StateAttribute < LState .
  extending ENVIRONMENT . extending STORE . extending CONTINUATION .
  op empty : -> LState .
  op _||_ : LState LState -> LState [assoc comm id: empty] .
  op k : Continuation -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
  op t : LState -> StateAttribute .
  op e : Env -> StateAttribute .
  op x : Continuation -> StateAttribute .
endfm
```

"`k`" wraps a continuation, "`n`" keeps the current free memory location, "`m`" the store, or "memory", "`t`" the state of a thread, "`e`" the execution environment of a thread, so it will be part of the state of a thread, and "`x`" a continuation of exceptions that will also be part of a thread's state. A typical state of a program in our language will have the following structure,

```
t(k(...) || e(...) || x(...) || ...) || ... || t(k(...) || e(...) || x(...) || ...) ||
...
n(N) ||
m(...) ||
...
```

where the local states of one or more threads are wrapped as global state attributes using constructors `t(...)`, where `N` is a number for the next free location, and `m(...)` keeps the store. Other state attributes can be added as needed, both inside threads and at the top level. Indeed, we will add top state attributes storing the locks that are taken by threads, and thread local attributes stating how many times each lock is taken by that thread. An important aspect of our semantic definitions, facilitated by Maude's ACI-matching capabilities, is that programming language features will be defined modularly, by referring to only those attributes that are needed. As we can see below, most of the semantic axioms refer to only one continuation! This way, one can add new features requiring new state attributes, *without having to change the already existing semantic definitions*. Moreover, equations and/or rules can be applied concurrently, thus increasing the efficiency of our interpreters and tools.

3.3 Defining the Semantics

The general intuition underlying CPS language definitions is that *control-contexts become data-contexts*, so they are manipulated like any other piece of data. Each continuation contains a sequence of execution obligations, which are stacked and processed accordingly. At each moment there is exactly one expression to be processed, namely the topmost expression in the stack. The following module gives CPS-semantics to generic expressions, that is, integers, identifiers, and lists of expressions. Integer values and several continuation items are needed:

```
mod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX . extending STATE .
  op int : Int -> Value .
  op _->_ : ExpList   Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op _->_ : Env       Continuation -> Continuation .
  var I : Int .  var K : Continuation .  var X : Id .  vars Env Env' : Env .
  var L : Location .  var V : Value .  var Vl : ValueList .
  var M : Store .  var E : Exp .  var El : ExpList .  var R : LState .
  eq k(I -> K) = k(int(I) -> K) .
  eq k(() -> K) = k((nil).ValueList -> K) .
  rl t(k(X -> K) || e([X,L] || Env) || R) || m([L,V] || M) =>
     t(k(V -> K) || e([X,L] || Env) || R) || m([L,V] || M) .
 ceq k((E,El) -> K) = k(E -> El -> K) if El =/= nil .
  eq k(V -> El -> K) = k(El -> V -> K) .
  eq k(Vl -> V -> K) = k(V,Vl -> K) .
  eq k(V -> Env -> K) || e(Env') = k(V -> K) || e(Env) .
endm
```

The definitions above deserve some discussion. A continuation of the form "`E -> K`" should be read and thought of as one "containing `E` followed by the rest of the computation/continuation `K`", and one of the form "`V -> K`" as one which "calculated `V` as the result of the previous expression at the top, but still has to process the computation/continuation `K`". Thus the first two equations above are clear. Similarly, a continuation "`Env -> K`" states that the current environment should be set to `Env`; this is needed in order to recover environments after processing bindings or function invocations. In fact, environments only need to be restored after a value is calculated in the modified environment; the last equation does exactly that. The other three equations process a list of expressions incrementally, returning a continuation containing a list of values at the top; note that, again, exactly one expression is processed at each moment.

The trickiest axiom in the above module is the rewriting rule, fetching the value associated to an identifier. It heavily exploits ACI matching and can intuitively be read as follows: if there is any thread whose continuation contains an identifier `X` at the top, whose environment maps `X` to a location `L`, and whose rest of resources `R` are not important, if `V` is the value associated to `L` in the store (note that the store is not part of any thread, because it is shared by all of them) then simply return `V`, the value associate to `X`, on top of the continuation; the rest of the computation `K` will know what to do with `V`. It is very important to note that this *must be a rule*! This is because the variable `X` may be shared by several threads, some of them potentially writing it via a variable assignment, so a variable read reflects a concurrent aspect of our programming language, whose behavior may depend upon the behavior of other threads.

The CPS semantics of arithmetic and boolean operators is straightforward: first place the operation to be performed in the continuation, then the expressions involved as a list in the desired order of evaluation (they can have side effects); after they are evaluated to a corresponding list of values, replace them by the result of the corresponding arithmetic or boolean operator. Note that a new type of value is needed for booleans:

```
fmod ARITHMETIC-EXP-SEMANTICS is extending ARITHMETIC-EXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  ops + - * / % : -> ContinuationItem .
  vars E E' : Exp .  vars I I' : Int .  var K : Continuation .
```

```
  eq k(E + E' -> K) = k((E,E') -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
*** -, *, /, % are defined similarly
endfm
fmod BOOLEAN-EXP-SEMANTICS is protecting BOOLEAN-EXP-SYNTAX .
  extending ARITHMETIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  ops == >= <= > < and or not : -> ContinuationItem .
  vars E E' : Exp . var K : Continuation . vars I I' : Int . vars B B' : Bool .
  eq k((E > E') -> K) = k((E,E') -> > -> K) .
  eq k((int(I),int(I')) -> > -> K) = k(bool(I > I') -> K) .
*** ==, >=, <=, < are defined similarly
  eq k((E and E') -> K) = k((E,E') -> and -> K) .
  eq k((bool(B),bool(B')) -> and -> K) = k(bool(B and B') -> K) .
*** 'or' and 'not' are defined similarly
endfm
```

The CPS semantics of conditionals "`if BE then E else E'`" is immediate: freeze `E` and `E'` in the current continuation `K` and then place `BE` on top and let it evaluate to a boolean value; then, depending upon the boolean value, unfreeze either `E` or `E'` and continue the computation. Note that a Maude "runtime error", i.e., a non-reducible term acting as a "core dump", will be obtained if `BE` does not evaluate to a boolean value. In fact, our programming language can be seen as a dynamically typed language; as shown in the lecture notes in [63], it is actually not hard to define static type checkers, but we do not discuss this aspect here:

```
fmod IF-SEMANTICS is protecting IF-SYNTAX . extending BOOLEAN-EXP-SEMANTICS .
  op if : Exp Exp -> ContinuationItem .
  vars BE E E' : Exp .  var K : Continuation .  var B : Bool .
  eq k((if BE then E else E') -> K) = k(BE -> if(E,E') -> K) .
  eq k(bool(B) -> if(E,E') -> K) = k(if B then E else E' fi -> K) .
endfm
```

Following the same CPS intuitions, the semantics of lists follows easily:

```
fmod LIST-SEMANTICS is protecting LIST-SYNTAX .
  extending BOOLEAN-EXP-SEMANTICS .
  ops list car cdr cons null? : -> ContinuationItem .
  var E E' : Exp .  var El : ExpList .  var K : Continuation .
  var V : Value .  var Vl : ValueList .  var Env : Env .
  eq k(list(El) -> K) = k(El -> list -> K) .   eq k(Vl -> list -> K) = k([Vl] -> K) .
  eq k(car(E) -> K) = k(E -> car -> K) .       eq k([V,Vl] -> car -> K) = k(V -> K) .
*** 'cdr', 'cons' and 'null' are defined similarly
endfm
```

Due to the simplifying rule in `FUNCTION-SYNTAX`, we only need to worry about giving semantics to functions of one argument, which can be either an identifier or "`()`". Since our language is statically scoped, we need to introduce a new value for *closures*, freezing the declaration environment of a function (1st equation). Function applications are defined as usual, by first evaluating the two expressions involved, the first being expected to evaluate to a closure, and then applying the first value to the second (a new continuation item, "`apply`" is needed). Two cases are distinguished here, when the argument of the function is the unit "`()`" or when it is an identifier `X`. In both cases the current environment is stored in the continuation, to be recovered later (after the evaluation of the body of the function) via the last equation in `GENERIC-EXP-SEMANTICS`, and the body of the function is evaluated in its declaration environment, that is frozen in the closure. When the function has an argument, a new location also needs to be created.

```
fmod FUNCTION-SEMANTICS is protecting FUNCTION-SYNTAX . extending GENERIC-EXP-SEMANTICS .
  op cl : Name Exp Env -> Value .
```

```
  op apply : -> ContinuationItem .
  var A : Name .  vars F E : Exp .  var K : Continuation .  vars Env Env' : Env .
  var X : Id .  var N : Nat .  var R : LState .  var V : Value .  var M : Store .
  eq k((fun A -> E) -> K) || e(Env) = k(cl(A,E,Env) -> K) || e(Env) .
  eq k(F E -> K) = k(F,E -> apply -> K) .
  eq k((cl((),E,Env), nil) -> apply -> K) || e(Env') = k(E -> Env' -> K) || e(Env) .
  eq t(k((cl(X,E,Env), V) -> apply -> K) || e(Env') || R) || n(N) || m(M) =
     t(k(E -> Env' -> K) || e(Env[X <- loc(N)]) || R) || n(N + 1) || m(M[loc(N) <- V]) .
endfm
```

LET and LETREC create new memory locations and change the execution environment. With the provided infrastructure, they are however quite easy to define. Note first that the desugaring translation in the module BINDING-SYNTAX reduces any list of bindings to a triple `(#,Xl,El)`, where `#` is the number of bindings, `Xl` is the list of identifiers to be bound, and `El` is the list of corresponding binding expressions. If `let (#,Xl,El) in E` is the current expression at the top of the continuation of a thread whose current environment is `Env` and whose rest of resources is `R`, and if `N` is the next available location (this is a global counter), then the CPS semantics works intuitively as follows: (1) freeze the current environment in the continuation, to be restored after the evaluation of `E`; (2) place `E` in the continuation; (3) generate `#` fresh locations and place in the continuation data-structure the information that these locations will be assigned to the identifiers in `Xl` at the appropriate moment, using an appropriate continuation item; (4) place the expressions `El` on top of the continuation; (5) once `El` is evaluated to a list of values `Vl`, they are stored at the new locations and the environment of the thread is modified accordingly, preparing for the evaluation of `E`; (6) after `E` is evaluated, the original environment will be restored, thanks to (1) and the last equation in GENERIC-EXP-SEMANTICS. All these technical steps can be compactly expressed with only two equations, again relying heavily on the ACI matching capabilities of Maude. Note that, despite their heavy use of memory, these equations do *not* need to be rules, because they can be executed deterministically regardless of the behavior of other threads. The fact that threads "compete" on the counter for the next available location `N` is immaterial, because there is no program whose behavior is influenced by which thread grabs `N` first.

```
fmod LET-SEMANTICS is protecting LET-SYNTAX . extending GENERIC-EXP-SEMANTICS .
  op `(_,_`) : IdList LocationList -> ContinuationItem .
  vars # N : Nat .  var Xl : IdList .  var El : ExpList .  var E : Exp .
  var K : Continuation .  var Env : Env .  var R : LState .  var M : Store .
  var Ll : LocationList .  var Vl : ValueList .
  eq t(k(let (#,Xl,El) in E -> K) || e(Env) || R) || n(N) =
     t(k(El -> (Xl,locs(N,#)) -> E -> Env -> K) || e(Env) || R) || n(N + #) .
  eq t(k(Vl -> (Xl,Ll) -> K) || e(Env) || R) || m(M) =
     t(k(K) || e(Env[Xl <- Ll]) || R) || m(M[Ll <- Vl]) .
endfm
```

The `let rec` construct gives a statically scoped language an enormous power by allowing one to define recursive functions. Semantically, the crucial difference between `let` and `let rec` is that the latter evaluates the bindings expressions `El` in the modified environment rather than in the original environment. Therefore, one first creates the new environment by mapping `Xl` to `#` fresh locations, then evaluates `El`, then stores their values at the new locations, then evaluates `E` and then restores the environment. This way, functions declared with `let rec` see each other's names in their closures, so they can call each other:

```
fmod LETREC-SEMANTICS is protecting LETREC-SYNTAX . extending GENERIC-EXP-SEMANTICS .
  op _->_ : LocationList Continuation -> Continuation .
  vars # N : Nat .  var Xl : IdList .  var El : ExpList .  var E : Exp .
  var K : Continuation .  var Env : Env .  var R : LState .  var M : Store .
  var Ll : LocationList .  var Vl : ValueList .
  eq t(k(let rec (#,Xl,El) in E -> K) || e(Env) || R) || n(N) =
     t(k(El -> locs(N,#) -> E -> Env -> K) || e(Env[Xl <- locs(N,#)]) || R) || n(N + #) .
  eq t(k(Vl -> Ll -> K) || R) || m(M) = t(k(K) || R) || m(M[Ll <- Vl]) .
endfm
```

So far, none of the language constructs had side effects. Variable assignments, `X := E`, evaluate `E` and store its value at the existing location of `X`. Therefore, `X` is expected to have been previously bound, otherwise a "runtime error" will be reported. It is very important that the actual writing of the value is performed using a rewriting rule, not an equation! This is because variable writing is a concurrent action, potentially influencing the execution of other threads that may read that variable. To distinguish this concurrent value writing from the value writing that occurred as part of the semantics of `let rec`, defined using the last equation in the module `LETREC-SEMANTICS`, we use a different continuation constructor for placing a location on a continuation ("`L => K`"):

```
mod ASSIGNMENT-SEMANTICS is extending ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .  var K : Continuation .
  var L : Location .  var V : Value .  var M : Store .  var R : LState .
  op _=>_ : Location Continuation -> Continuation .
  eq k((X := E) -> K) || e([X,L] || Env) = k(E -> L => noValue -> K) || e([X,L] || Env) .
  rl t(k(V -> L => K) || R) || m(M) => t(k(K) || R) || m(M[L <- V]) .
endm
```

Blocks are quite straightforward: the semicolon-separated expressions are evaluated in order and the result of the evaluation of the block is the value of the last expression; the values of the other expressions except the last one are ignored. Therefore, the expressions in a block are used for their side effects:

```
fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX . extending GENERIC-EXP-SEMANTICS .
  op ignore : -> ContinuationItem .
  var E : Exp . var Eb : ExpBlock . var K : Continuation . var V : Value .
  eq k({E} -> K) = k(E -> K) .
  eq k({E ; Eb} -> K) = k(E -> ignore -> {Eb} -> K) .
  eq k(V -> ignore -> K) = k(K) .
endfm
```

There is nothing special in the CPS semantics of loops: the body of the loop followed by its condition are placed on top of the continuation at each iteration, and the loop is terminated when the condition becomes false. The evaluation of loops returns no value, so loops are also used just for their side effects:

```
fmod LOOP-SEMANTICS is extending LOOP-SYNTAX . extending BOOLEAN-EXP-SEMANTICS .
  op while(_,_) : Exp Exp -> ContinuationItem .
  vars BE E : Exp .  var Vl : ValueList .  var K : Continuation .
  eq k((while BE E) -> K) = k(BE -> while(BE,E) -> K) .
  eq k((Vl,bool(true)) -> while(BE,E) -> K) = k((E,BE) -> while(BE,E) -> K) .
  eq k((Vl,bool(false)) -> while(BE,E) -> K) = k(noValue -> K) .
endfm
```

We next define the semantics of exceptions. Whenever an expression of the form `try E catch E'` is encountered, `E'` is first evaluated. Then `E` is evaluated; if `throw(E'')` is encountered during the evaluation of `E`, then the entire control context within `E` is immediately discarded and the value of `E''` is passed to

`E'`, which was previously supposed to evaluate to a function; if no exception is thrown during the evaluation of `E`, then `E'` is discarded and the value of `E` is returned as the value of `try E catch E'`. An interesting technicality here is that the above mechanism can be elegantly implemented by maintaining an additional continuation within each thread, wrapped within "`x(...)`", freezing and stacking the control contexts, i.e., the continuations, at the times when the `try E catch E'` expressions are encountered (note that these can be nested):

```
fmod EXCEPTION-SEMANTICS is protecting EXCEPTION-SYNTAX . extending FUNCTION-SEMANTICS .
  op try : Exp -> ContinuationItem .
  op popx : -> Continuation .
  op `(_,_`) : Value Continuation -> ContinuationItem .
  op throw : -> ContinuationItem .
  vars E E' : Exp .  vars K K' EX : Continuation .  vars V V' : Value .
  eq k(try E catch E' -> K) = k(E' -> try(E) -> K) .
  eq k(V' -> try(E) -> K) || x(EX) = k(E -> popx) || x((V',K) -> EX) .
  eq k(V -> popx) || x((V',K) -> EX) = k(V -> K) || x(EX) .
  eq k(throw(E) -> K) = k(E -> throw -> K) .
  eq k(V -> throw -> K') || x((V',K) -> EX) = k((V',V) -> apply -> K) || x(EX) .
endfm
```

The only feature left to define is threads. In what follows we assume that one already has a definition for sets of integers with membership, `INT-SET`, and one for sets of pairs of integers, `COUNTER-SET`. Both of these are trivial to define, so we do not discuss them here. The former will be used to store all the synchronization objects, or *locks*, that are already acquired, and the latter to store how many times a thread has acquired a lock, so that it knows how many times it needs to release it. These are wrapped as program state attributes:

```
mod THREAD-SEMANTICS is protecting THREAD-SYNTAX .  extending GENERIC-EXP-SEMANTICS .
  protecting INT-SET .  protecting COUNTER-SET .
  op c : CounterSet -> StateAttribute .
  op b : IntSet -> StateAttribute .
```

A new type of value is needed, namely one for locks. A lock is just an integer, which is wrapped with `lockv` to keep it distinct from other integer values:

```
  op lockv : Int -> Value .
```

Newly created threads are executed for their side effects. At the end of their execution, threads release their locks and kill themselves. Therefore, we introduce a new continuation, `die`, to distinguish the termination of a thread from the termination of the main program. The creation of a new thread is a no value operation. The new thread inherits the execution environment from its parent:

```
  op die : -> Continuation .
  ops lock acquire release : -> ContinuationItem .
  var E : Exp . var K : Continuation . var Env : Env . var R : LState . var I : Int .
  var V : Value . var Cs : CounterSet . var Is : IntSet . var N : Nat . var Nz : NzNat .
  eq t(k(spawn(E) -> K) || e(Env) || R) = t(k(noValue -> K) || e(Env) || R) ||
       t(k(E -> die) || e(Env) || x(stop) || c(empty)) .
  eq t(k(V -> die) || c([I,N] || Cs) || R) || b(I || Is) =
     t(k(V -> die) || c(Cs) || R) || b(Is) .
  eq t(k(V -> die) || c(empty) || R) = empty .
```

Locks are values which can be handled like any other values in the language. In particular, they can be passed to and returned as results of functions; `lock(E)` evaluates `E` to an integer value `I` and then generates the value `lockv(I)`:

```
  eq k(lock(E) -> K) = k(E -> lock -> K) .
  eq k(int(I) -> lock -> K) = k(lockv(I) -> K) .
```

Acquiring a lock needs to distinguish two cases. If the current thread already has the lock, reflected in the fact that it has a counter associated to that lock, then it just needs to increment that counter. This operation is not influenced by, and does not influence, the execution of other threads, so it can be defined using an ordinary equation. The other case, when a thread wants to acquire a lock which it does not hold already, needs to be a rewriting rule for obvious reasons: the execution of other threads may be influenced, so the global behavior of the program may be influenced. Once the new lock is taken, a thread local counter is created and initialized to 0, and the lock is declared "busy" in `b(...)`. This rule is conditional, in that the lock can be acquired only if it is not busy:

```
  eq k(acquire(E) -> K) = k(E -> acquire -> K) .
  eq k(lockv(I) -> acquire -> K) || c([I, N] || Cs) =
     k(noValue -> K) || c([I, N + 1] || Cs) .
 crl t(k(lockv(I) -> acquire -> K) || c(Cs) || R) || b(Is) =>
     t(k(noValue -> K) || c([I, 0] || Cs) || R) || b(I || Is)  if not(I in Is) .
```

Dually, releasing a lock also involves two cases. However, both of these can be safely defined with equations, because threads do not need to compete on releasing locks:

```
  eq k(release(E) -> K) = k(E -> release -> K) .
  eq k(lockv(I) -> release -> K) || c([I, Nz] || Cs) =
     k(noValue -> K) || c([I, Nz - 1] || Cs) .
  eq t(k(lockv(I) -> release -> K) || c([I, 0] || Cs) || R) || b(I || Is) =
     t(k(noValue -> K) || c(Cs) || R) || b(Is) .
endm
```

All the features of our programming language have been given CPS rewriting logic semantics, so we can now put all the semantic specifications together and complete the definition of our language:

```
fmod LANGUAGE-SEMANTICS is extending ARITHMETIC-EXP-SEMANTICS .
  extending BOOLEAN-EXP-SEMANTICS .  extending IF-SEMANTICS .
  extending LET-SEMANTICS .          extending LETREC-SEMANTICS .
  extending FUNCTION-SEMANTICS .     extending LIST-SEMANTICS .
  extending ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .        extending LOOP-SEMANTICS .
  extending EXCEPTION-SEMANTICS .    extending THREAD-SEMANTICS .
  op eval : Exp -> [Value] .
  op [_] : LState -> [Value] [strat(1 0)] .
  var E : Exp .  vars R S : LState .  var V : Value .
  eq eval(E) = [t(k(E -> stop) || e(empty) || x(stop) || c(empty)) ||
                n(0) || m(empty) || b(empty)] .
  eq [t(k(V -> stop) || R) || S] = V .
endfm
```

The main operator that enables all the semantic definitions above is `eval`, which may or may not return a proper value. As the definition of the auxiliary operator `[_]` shows, `eval` returns a proper value if and only if the original thread (the only one whose continuation is built on top of `stop`) evaluates to a proper value `V`. The definition of `eval` above also shows the various state attributes involved, as well as their nesting and grouping: the thread state attribute, `t(...)`, includes a continuation (`k`), an environment (`e`), an exception continuation (`x`), and a lock counter set (`c`); the other global state attributes, laying at the same top level as the thread attributes, are a counter for the next free location (`n`), a "memory" wrapping a mapping from locations to values (`m`), and a set of "busy" locks (`b`).

3.4 Getting an Interpreter for Free

Since Maude can efficiently execute rewriting logic specifications, an immediate benefit of defining the semantics of a programming language in rewriting logic is that we obtain an *interpreter* for that language with no extra effort. All what we have to do is to "rewrite" terms of the form `eval(E)`, which should reduce to values. For example, the evaluation of the following factorial program

```
rew eval(
  let rec f n =
    if n == 0 then 1 else n * f(n - 1)
  in f 100
) .
```

takes 5151 rewrites and terminates in 64ms[8]. with the following result:

```
result Value: int(93326215443944152681699238856266700490715968264381621468592963895217599
99322991560894146397615651828625369792082722375825118521091686400000000000000000000000000)
```

The following recursive program calculating the product of elements in a list is indeed evaluated to `int(0)` (in 716 rewrites). This program is "inefficient" because the product function returns normally from its recursive calls when a 0 is encountered, which can be quite time consuming in many situations:

```
rew eval(
  let rec p l =
    if null?(l) then 1 else if car(l) == 0 then 0 else car(l) * (p cdr(l))
  in p list(1,2,3,4,5,6,7,8,9,0,10,11,12,13,14,15,16,17,18,19,20)
) .
```

Since our language has exceptions, a better version of the same program (reducing in 675 rewrites) is one which throws an exception when a 0 is encountered, thus exiting all the recursive calls at once:

```
rew eval(
  let p l = try let rec a l =
                    if null?(l) then 1
                    else if car(l) == 0 then throw 0 else car(l) * (a cdr(l))
                  in a l
                catch fun x -> x
  in p list(1,2,3,4,5,6,7,8,9,0,10,11,12,13,14,15,16,17,18,19,20)
) .
```

To illustrate the imperative features of our language, let us consider Collatz' conjecture, stating that the procedure, dividing `n` by 2 if it is even and multiplying it by 3 and adding 1 if it is `odd`, eventually reaches 1 for any `n`). For our particular `n` below, it takes 73284 rewrites in 0.3s to evaluate to `int(813)`:

```
rew eval(
  let n = 21342423543653426527423676545 and c = 0
  in {while n > 1 {
          if 2 * (n / 2) == n then n := n / 2 else n := 3 * n + 1 ;
          c := c + 1
        } ;
      c }
) .
```

Let us next illustrate some concurrent aspects of our language. The following program spawns a thread that assigns 1 to `a` and then recursively increments a counter `c` until `a` becomes indeed 1. Any possible value for the counter can be obtained, depending upon when the spawned thread is scheduled for execution:

[8] All the performance results in this paper were measured on a 2.4GHz PC.

```
rew eval(
  let a = 0 and c = 0 in {
    spawn(a := 1) ;
    let rec f() = if a == 1 then c else {c := c + 1 ; f()} in f()
  }
) .
```

We are currently letting Maude schedule the execution of threads based on its internal default scheduler for applications of rewrite rules, which in the example above leads to an answer `int(0)`. Note, however, that one can also use Maude's fair rewrite command `frew` instead of `rew`, or one can even define one's own scheduler using Maude's meta-level capabilities. Even though we will not discuss thread scheduling aspects in this paper, in the next section we will show how one can use Maude's `search` capability to find executions leading to other possible results for the program above, for example `int(10)`.

An inherent problem in multithreaded languages is that several threads may access the same location at the same time, and if at least one of these accesses is a write this can lead to *dataraces*. The following program contains a datarace:

```
rew eval(
  let x = 0 in {
    spawn(x := x + 1) ;
    spawn(x := x + 1) ;
    x
  }
) .
```

Maude's default scheduler happens to schedule the two spawned threads above such that no datarace occurs, the reported answer being `int(2)`. However, under different thread interleavings the reported value of `x` can also be 0 or even 1. The latter reflects the datarace phenomenon: both threads read the value of `x` before any of them writes it, and then they both write the incremented value. Using `search`, we show in the next section that both `int(0)` and `int(1)` can be valid results of the program above. Thread synchronization mechanisms are necessary in order to avoid dataraces. We use locks for synchronization in our language. For example, the following program is datarace free, because each thread acquires the lock `lock(1)` before accessing its critical region. Note, however, that the final result of this program is still non-deterministic (can be either 2 or -1):

```
rew eval(
  let a = 1 and b = 1 and x = 0 and l = lock(1) in {
    spawn {acquire l ; x := x + 1 ; release l ; a := 0} ;
    spawn {acquire l ; x := x + 1 ; release l ; b := 0} ;
    if (a == 0) and (b == 0) then x else -1
  }
) .
```

3.5 Specifying Java and the JVM

The language presented above was selected and designed to be as simple as possible, yet including a substantial range of features, such as high-order and imperative features, static scoping, recursion, exceptions and concurrency. However, we are actively using the rewriting logic semantics approach to formally define different programming paradigms and large fragments of several languages, including Scheme, OCaml, ML, Pascal, Java, and JVM, several of them covered in a programming language design course at UIUC [63].

Java has been recently defined at UIUC by Feng Chen in about three weeks, using a CPS semantics as above, with 600 equations and 15 rewrite rules. Azadeh Farzan has developed a more direct rewriting logic specification for the JVM, not based on continuations, specifying about 150 out of 250 bytecode instructions with around 300 equations and 40 rewrite rules. The continuations-based style used in this paper should be regarded as just a definitional methodology, which may not be appropriate for some languages, especially for lower level ones. Both the Java and the JVM specifications include multithreading, inheritance, polymorphism, object references, and dynamic object allocation. We do not support native methods nor many of the Java built-in libraries at present. The definition of Java follows closely the style used to define our sample language above, with states consisting of multisets of potentially nested state attributes. A new type of value is introduced for objects, wrapping also the name of the class that the object is an instance of, which is necessary in order to have access to that object's methods. The essential difference in the definitional styles of Java and the JVM is that the latter follows the object paradigm of Maude [41], considering objects also as part of the state multiset structure; and method calls are translated into messages that objects can send to each other, by placing them into the multiset state as well. Rewrites (with rewrite rules and equations) in this multiset model the changes in the state of the JVM.

4 Formal Analysis of Concurrent Programs

Specifying formally the rewriting logic semantics of a programming language in Maude, besides providing an increased understanding of all the details underlying a language design, also yields a prototype interpreter for free. Furthermore, a solid foundational framework for program analysis is obtained. It is conceptually meaningless to speak about rigorous verification of programs without a formal definition of the semantics of that language. Once a definition of a language is given in Maude, thanks to generic analysis tools for rewriting logic specifications that are efficiently implemented and currently provided as part of the Maude system, we additionally get the following important 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]) that can be used to semi-automatically prove programs correct.

We only focus on the first two items in this paper, because they are entirely automatic (except of course for the need to equationally define the atomic predicates of interest in temporal logic formulas).

4.1 Search

We have seen several examples where concurrent programs can have quite nondeterministic behaviors due to many possible thread interleavings, some of them leading to undesired behaviors, e.g., dataraces, due to lack of synchronization.

Using Maude's `search` command, one can search a potentially infinite state space for behaviors of interest. Since such a search is performed in a breadth-first manner, if any safety violation exists then it will eventually be found, i.e., this is a semi-decision procedure for finding such errors. For example, the following two-threaded program which evaluates to 0 in Maude under its default scheduling, can be shown to evaluate to any possible integer value. It takes 11ms to find an interleaving that leads to `int(10)`, after exploring 108 states:

```
search [1] eval( let a = 0 and c = 0 in {
                   spawn(a := 1) ;
                   let rec f() = if a == 1 then c else {c := c + 1 ; f()} in f()
                 } ) =>* int(10) .
```

One can show that the poorly synchronized program in Section 3.4 has a datarace,

```
search [1] eval( let x = 0 in {
                   spawn(x := x + 1) ;
                   spawn(x := x + 1) ;
                   x
                 } ) =>+ int(1) .
```

and also that the properly synchronized version of it is datarace free:

```
search [1] eval( let a = 1 and b = 1 and x = 0 and l = lock(1) in {
                   spawn {acquire l ; x := x + 1 ; release l ; a := 0} ;
                   spawn {acquire l ; x := x + 1 ; release l ; b := 0} ;
                   if (a == 0) and (b == 0) then x else -1
                 } ) =>+ int(1) .
```

The above returns "`No solution`", after exploring all 90 possible states in 23ms. If one wants to see the state space generated by the previous `search` command, one can type the command "`show search graph`". An interesting example showing that dataraces can be arbitrarily dangerous was proposed by J. Moore [53], where two threads performing the assignment "`c := c + c`" for some shared variable `c`, can lead to any possible integer value for `c`. The following shows how one can test whether the value `25` can be reached. It takes Maude about 1s to explore 4696 states and find a possible interleaving leading to the final result 25:

```
search [1] eval( let rec c = 1 and f() = {c := c + c ; f()} in {
                   spawn f() ;
                   spawn f() ;
                   c
                 } ) =>! int(25) .
```

4.2 Model Checking

When the state space of a concurrent program is finite, one can exhaustively analyze all its possible executions and check them against temporal logic properties. Currently, Maude provides a builtin explicit-state model checker for linear temporal logic (LTL) comparable in speed to SPIN [23], which can be easily used to model check programs once a programming language semantics is defined as a rewriting logic specification. The module `MODEL-CHECKER`, part of Maude's distribution, exports sorts `State`, `Prop`, and a binary "satisfaction" operator `_|=_ : State Prop -> Bool`. In order to define temporal properties to model-check, the user has to first define state predicates using the satisfaction operation.

To exemplify this analysis technique, we next consider the classical *dining philosophers* problem. The property of interest in this example is that the program terminates, so we only need one state predicate, `terminates`, which holds whenever a proper value is obtained as a result of the execution of the program (note that `eval` may not always return a proper value; its result was the kind `[Value]`):

```
fmod CHECK is extending MODEL-CHECKER .  extending LANGUAGE-SEMANTICS .
  subsort Value < State .
  op terminates : -> Prop .
  eq V:Value |= terminates = true .
endfm
```

We can model check a five dining philosophers program as follows:

```
red modelCheck(eval( let n = 5 and i = 1 and
                         f x = { acquire lock(x) ; acquire lock(x + 1) ;
                                 --- eat
                                 release lock(x + 1) ; release lock(x) }
                     in { while i < n
                            { spawn(f i) ; i := i + 1 } ;
                          acquire lock(n) ; acquire lock(1) ;
                          --- eat
                          release lock(1) ; release lock(n) } ), <> terminates) .
```

Maude's model checker detects the deadlock and returns a counterexample trace in about 0.5s. If one fixes this program to avoid deadlocks, for example as follows:

```
red modelCheck(eval( let n = 5 and i = 1 and
                         f x = 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) }
                     in { while i < n
                           { spawn(f i) ; i := i + 1 } ;
                          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) } } ), <> terminates) .
```

then the model-checker analyzes the entire state space and returns `true`, meaning that the program will terminate for any possible thread interleaving.

4.3 Formal Analysis of Java Multithreaded Programs

In joint work with Azadeh Farzan and Feng Cheng, we are using Maude to develop JavaFAN (Java Formal ANalyzer) [26, 25], a tool in which Java and JVM code can be executed and formally analyzed. JavaFAN is based on Maude rewriting logic specifications of Java and JVM (see Section 3.5). Since JavaFAN is intended to be a Java analysis tool rather than a programming language design platform, we have put a special emphasis on its *efficiency*. When several ways to give semantics to a feature were possible, we have selected the one which performed better on our benchmarks, instead of the mathematically simplest one. In this section we discuss JavaFAN and some of the experiments that we performed with it. They support the claim that the rewriting logic approach to formal semantics of programming languages presented in this paper is not only a clean theoretical model unifying SOS and equational semantics, but also a potentially powerful practical framework for developing software analysis tools.

Figure 1 presents the architecture of JavaFAN. The *user interface* module hides Maude behind a user-friendly environment. It also plays the role of a

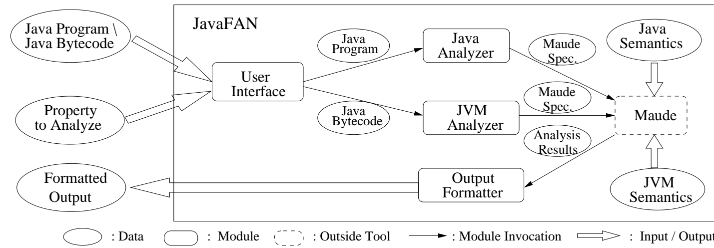


Fig. 1. Architecture of JavaFAN.

dispatcher, sending the Java source or the bytecode to Java or JVM analyzers, respectively. The analyzers wrap the input programs into Maude modules and invoke Maude, which analyzes the code based on the formal specifications of the Java language and of the JVM. The output formatter collects the output of Maude, transforms it into a user-readable format, and sends it to the user.

We next discuss some of the examples analyzed with JavaFAN and comparisons to other similar tools. The *Remote Agent* (RA) is a spacecraft controller, part NASA's Deep Space 1 shuttle, that deadlocked 96 million miles from Earth due to a datarace. This example has been extensively studied in [31, 32]. JavaFAN's search found the deadlock in 0.1 seconds at the source code level and in 0.3 seconds at the bytecode level, while the tool in [60] finds it in more than 2 seconds. Another comparison with [60] was done on a 2 stage pipeline code, each stage executing as a separate thread, against a property taken from [60]. JavaFAN model checks the property in 17 minutes, while the tool in [60], without partial order reduction optimizations[9], does it in more than 100 minutes. JavaFAN can detect the deadlock for up to 9 philosophers. Other Java model checkers, with support for heuristics and abstraction techniques such as Java PathFinder (JPF) [87, 10, 33], can do larger numbers. If the deadlock potential is removed, like in Section 4.2, thus diminishing the role of heuristics, then JavaFAN can prove the program deadlock-free for up to 7 philosophers, while JPF cannot deal with 4 philosophers (on the same program). All these examples as well as the JavaFAN system are available on the web [24].

4.4 Performance of the Formal Analysis Tools

There are two reasons for the efficiency of the formal analysis tools for languages whose rewriting logic semantics is given in Maude, and in particular for which JavaFAN compares favorably with more conventional Java analysis tools:

1. The high performance of Maude for execution, search, and model checking;
2. The optimized equational and rule definitions.

Maude's rewriting engine is highly optimized and can perform millions of rewrite steps per second, while its model checker is comparable in speed with SPIN [23]. In addition to these, we have used performance-enhancing specification techniques, including: expressing as equations the semantics of all deterministic computations, and as rules only concurrent computations (since rewriting

[9] JavaFAN is currently just a brute force, unoptimized explicit state model checker.

happens *modulo* equations, only rules contribute to state space size); favoring *unconditional* equations and rules over less efficient conditional versions; and using a continuation passing style in semantic equations.

5 SOS and Equational Semantics Revisited

Now that rewriting logic semantics has been explained and has been illustrated in detail, we take a second look at how equational semantics and SOS are unified within rewriting logic semantics. We also explain how their respective limitations are overcome within this broader semantic framework.

5.1 Unification of Equational Semantics

If R is empty in a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, then ϕ is irrelevant and $\mathcal{R}$ becomes an *equational theory*, and the initial model $T_{Reach(\mathcal{R})}$ becomes in essence the *initial algebra* $T_{\Sigma/E}$. Therefore, equational logic is a *sublogic* of rewriting logic, and *initial algebra semantics* is a special case of rewriting logic's initial model semantics. That is, equational semantics is a *special case* of rewriting logic semantics, namely the case when $R = \varnothing$. Higher-order semantic equations can be integrated in two alternative ways. On the one hand, we can make everything first-order by means of an *explicit substitution* calculus or the use of combinators. On the other hand, we can embed higher-order semantic equations within a higher-order version of rewriting logic such as Stehr's open calculus of constructions (OCC) [70]. Either way, since OCC and many other higher-order calculi can be faithfully represented in first-order rewriting logic [71, 70], it is possible to execute such definitions in a rewriting logic language such as Maude.

Integrating equational semantics within rewriting logic makes the limitations in handling concurrency mentioned in Section 1.1 disappear, since all deterministic computations of a language can still be specified by equations, but the means missing in equational semantics to properly handle concurrency are now provided by rewrite rules. Furthermore, the extension from equational logic to rewriting logic is conservative and all the good proof- and model-theoretic properties are preserved in the extension. This leaves us with the pending issue of modularity, which is discussed in Section 5.3.

5.2 Unification of SOS and Reduction Semantics

SOS can also be integrated within rewriting logic. This has been understood from the early stages of rewriting logic [40, 47, 39], and has led to several implementations of SOS definitions [8, 80]. 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*. There is however an important difference 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* transition. Instead, because of **Reflexivity** and **Transitivity**, a rewriting

logic sequent $P \longrightarrow Q$ may involve many rewrite steps; furthermore, because of the **Congruence**, such steps may correspond to rewriting subterms.

Since the conditions in a conditional rewrite rule may involve many rewrite steps, whereas the transitions in the condition of an SOS rule are one-step transitions, in order to faithfully represent an SOS rule we have somehow to "dumb down" the rewriting logic inference system. Of course we do not want to actually change rewriting logic's inference rules: we just want to *get the effect* of such a change, so that in fact only the **Replacement** rule is used. This can be achieved by representing an SOS specification as a suitable rewrite theory that, due to its construction, precludes the application of the other inference rules in the logic. We explain how this can be done for an SOS specification consisting of unlabeled SOS rules of the general form described above. We can think of such an SOS specification as a pair $\mathcal{S} = (\Sigma, R)$, where Σ is a many-sorted signature, and the rules R are of the general form described above, where the Ps and Qs are Σ-terms having the same sort whenever they appear in the same transition. The SOS rules are then applied to substitution instances of the patterns appearing in each rule in the usual SOS way. The corresponding rewrite theory representing $\mathcal{S}$ is denoted $\widehat{\mathcal{S}}$ and is $\widehat{\mathcal{S}} = (\widehat{\Sigma}, OP, \phi, \widehat{R})$, where:

- $\widehat{\Sigma}$ is the MEL signature obtained from Σ by:
  - adding a kind $[s]$ for each sort s in Σ, so that the set of sorts for kind $[s]$ is the singleton set $\{s\}$
  - adding for each $f : s_1 \dots s_n \longrightarrow s$ in Σ an operator $f : [s_1] \dots [s_n] \longrightarrow [s]$
  - adding for each kind [s] two operators $[_] : [s] \longrightarrow [s]$ and $\langle _ \rangle : [s] \longrightarrow [s]$.
- OP is the set of axioms associating to each operator $f : s_1 \dots s_n \longrightarrow s$ in Σ the membership $\forall (x_1 : s_1, \dots, x_n : s_n)\ f(x_1, \dots, x_n) : s$, so that terms of the old sorts in Σ remain well-sorted in $\widehat{\Sigma}$
- ϕ declares all arguments in all operators in $\widehat{\Sigma}$ *frozen*, and
- $\widehat{R}$ has for each SOS rule in R a corresponding rewrite rule

$$[Q] \longrightarrow \langle Q' \rangle \quad \text{if} \quad [P_1] \longrightarrow \langle P_1' \rangle \ \wedge\ \dots\ \wedge\ [P_n] \longrightarrow \langle P_n' \rangle,$$

The key result is then the following lemma, that we state without proof:

Lemma 1. *For any ground Σ-terms t, t' of the same sort, we have*

$$\mathcal{S} \vdash_{SOS} t \longrightarrow t' \quad \Leftrightarrow \quad \widehat{\mathcal{S}} \vdash_{RWL} [t] \longrightarrow \langle t' \rangle$$

where $\vdash_{SOS}$ and $\vdash_{RWL}$ denote the SOS and rewriting logic inference systems.

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) [56, 57, 54], has managed to neatly pack all the various pieces of semantic information usually scattered throughout a standard SOS rule *inside rule labels*, 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 [46] defines a faithful representation of an MSOS specification $\mathcal{S}$ as a corresponding rewrite theory $\tau(\mathcal{S})$, provided the MSOS rules in $\mathcal{S}$ are in a suitable normal form. Such MSOS rules do in fact have labels that include any desired semantic information, and can have equational side conditions. A semantic equivalence result similar to the above lemma holds between transitions in $\mathcal{S}$ and one-step rewrites in $\tau(\mathcal{S})$ [46]. This shows the MSOS specifications are faithfully represented by their rewriting logic translations.

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 adopted to give semantics to different concurrent calculi and programming languages (see [3, 50] for two early references). Since the 1990 San Miniato Workshop on Concurrency, where both the Cham and rewriting logic were presented [21], it has been clearly understood that these are two closely related formalisms, so that each Cham can be naturally seen as a rewrite theory (see [40] Section 5.3.3, and [3]). 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 $\mathcal{R} = (\Sigma, A, \phi, R)$, where A consists of *structural axioms*, e.g., associativity and commutativity of multiset union for the Cham[10], and R is typically 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.

Limitations of SOS similar to those pointed out in Section 1.1 were also clearly perceived by Berry and Boudol, so that the Cham is proposed not as a variant of SOS, but as an *alternative semantic framework* (see [3], Section 2.3). Indeed, an important theme is overcoming the *rigidity of syntax*, forcing traditional SOS to express communication in a centralized, interleaving way, whereas the use of associativity and commutativity and the *locality* of rewrite rules allows a more natural expression of local concurrent interactions. On this point rewriting logic semantics and reduction semantics are in full agreement. Four further advantages added by rewriting logic semantics to overcome other limitations of SOS mentioned in Section 1.1 are: (i) the existence of a model-theoretic semantics having initial models, that smoothly integrates the model theory of algebraic semantics as a special case and serves as a basis for inductive and temporal logic reasoning; (ii) the more general use of equations not only as *structural axioms* A (e.g., AC of multiset union for the Cham) but also as *semantic equations* E_0 that are Church-Rosser modulo A, so that in general we have $E = E_0 \cup A$; (iii) allowing *conditional rewrite rules* which permits a natural integration of SOS within rewriting logic; and (iv) the existence of high-performance implementations supporting both execution and formal analysis. This brings us to the last limitation mentioned in Section 1.1 for both equational semantics and SOS, namely modularity.

[10] As pointed out in [40], the Cham's heating and cooling rules and the airlock rule could also be seen as equations and could be made part of the set A.

5.3 Modularity

Both equational semantics and SOS are notoriously *unmodular*. That is, when a new kind of feature is added to the existing formal specification of a language's semantics, it is often necessary to introduce *extensive redefinitions* in the earlier specification. One would of course like to be able to define the semantics of each feature in a modular way *once and for all*, but this is easier said than done.

Rewriting logic as such does not solve the modularity problem. After all, equational definitions remain untouched when embedded in rewriting logic, and SOS definitions, except for the technicality of restricting rewrites to one step in conditions, are represented by quite similar conditional rewrite rules. Therefore, if the specifications were unmodular beforehand, it is unreasonable to expect that they will magically become modular when viewed as rewrite theories. Something else is needed, namely a *modular specification methodology*.

In this regard, the already mentioned work of Mosses on MSOS [56, 57, 54] is very relevant and important, because it has given a simple and elegant solution to the SOS modularity problem. Stimulated by Mosses' work, the first author, in joint work with Christiano Braga, has investigated a methodology to make rewriting logic definitions of programming languages modular. The results of this research are reported in [46], and case studies showing the usefulness of these modularity techniques for specific language extensions are presented in [9]. In particular, since equational logic is a sublogic of rewriting logic, the modular methodology proposed in [46] specializes in a straightforward way to a new modular specification methodology for algebraic semantics. The two key ideas in [46] are the systematic use of ACI matching to make semantic definitions impervious to the later addition of new semantic entities, and the systematic use of *abstract interfaces* to hide the internal representations of semantic entities (for example a store) so that such internal representations can be changed in a language extension without a need to redefine the earlier semantic rules.

This methodology has influenced the specification style used in Section 3, even though the methodology in [46] is not followed literally. One limitation mentioned in [46] is the somewhat rigid style imposed by assuming configurations consisting of a program text and a record of semantic entities, which forces an interleaving semantics alien in spirit to rewriting logic's true concurrency semantics. One can therefore regard the specification style illustrated in Section 3 as a snapshot of our current steps towards a truly concurrent modular specification methodology, a topic that we hope to develop fully in the near future.

6 Concluding Remarks

We have introduced rewriting logic, have explained its proof theory and its model theoretic semantics, and have shown how it unifies both equational semantics and SOS within a common semantic framework. We have also explained how reduction semantics can be regarded as a special case of rewriting logic semantics. Furthermore, we have shown how rewriting logic semantic definitions written in a language like Maude can be used to get efficient program analysis tools, and

have illustrated this by means of a substantial Caml-like language specification. The unification of equational semantics and SOS achieved this way combines the best features of these approaches and has the following advantages:

- a rewrite theory $\mathcal{R}$ has an initial model semantics given by $\mathcal{T}_{\mathcal{R}}$, and a proof-theoretic operational semantics given by rewrite proofs; furthermore, by the Completeness Theorem both semantics *agree*
- the initial model $\mathcal{T}_{\mathcal{R}}$ provides the mathematical basis for formal reasoning and theorem proving in first- and higher-order inductive theorem proving and in temporal logic deduction
- rewriting logic provides a *crucial distinction* between semantic *equations* E and semantic *rules* R, that is, a distinction between deterministic and concurrent computation not available in either equational semantics or SOS
- such a distinction is key not only conceptually, but also for efficiency reasons of drastically collapsing the state space
- rewriting logic has a *true concurrency* semantics, more natural than an interleaving semantics when defining concurrent languages with features such as distribution, asynchrony, and mobility
- when specified in languages like Maude, semantic definitions can be turned into efficient interpreters and program analysis tools for free
- when developed according to appropriate methodological principles, rewriting logic semantic definitions become *modular* and are easily extensible without any need for changes in earlier semantic rules.

An important aspect of the rewriting logic semantics we propose is the flexibility of choosing the desired *level of abstraction* at will when giving semantic definitions. Such a level of abstraction may be different for different modeling and analysis purposes, and can be easily changed as explained below. The point is that in a rewrite theory (Σ, E, ϕ, R), rewriting with the rules R happens *modulo* the equations in E. Therefore, the more semantic definitions we express as equations the more *abstract* our semantics becomes. Abstraction has important advantages for making search and model checking efficient, but changes what is *observable* in the model. In this sense, the Caml-like language specification in Section 3 is quite abstract; in fact, it has only three rewrite rules, with all other axioms given as equations. It is indeed possible to observe all global memory changes, since these are all expressed with rules, but some other aspects of the computation may not be observable at this level of abstraction. For example, nonterminating local sequential computations, such as a nonterminating function call or while loop, will remain within the same equivalence class. This may even lead to starvation of other threads in an interpreter execution. Generally speaking, when observing a program's computation in a more fine-grained way becomes important, this can be easily done by *transforming some equations into rules*. For example, one may wish to specify all potentially nonterminating constructs with rules. The most fine-grained way possible is of course to transform *all equations* (except for structural axioms such as ACI) into rules. These transformations are easy to achieve, since they amount to very simple changes in the specification. In fact, one may wish to use different variants of a language's

specification, with certain semantic definitions specified as equations in one variant and as rules in another, because each variant may provide the best level of abstraction for a different set of purposes. The moral of the story is precisely that rewriting logic's distinction between equations and rules provides a useful "abstraction knob" by which we can fine tune a language's specification to best handle specific formal analysis purposes. There are a number of open research directions suggested by these ideas:

- for model checking scalability purposes it will be important to add techniques such as partial order reduction and predicate abstraction;
- besides search and model checking, using rewriting logic semantic definitions as a basis for *theorem proving* of program properties is also a direction that should be vigorously pursued; this semantics-based method is well-understood for equational semantics [29] and has been used quite successfully by other researchers in the analysis of Java programs using both PVS [36] and ACL2 language specifications [51]; in the context of Maude, its ITP tool [19] has been already used to certify state estimation programs automatically synthesized from formal specifications [65, 64] and also to verify sequential programs based on a language's semantic definition [45].
- rewriting is a simple and general model of computation, and rewriting-based semantic definitions already run quite fast on a language like Maude which is itself a semi-compiled interpreter; this suggests that, given an appropriate compilation technology for rewriting, one could directly compile programming languages into a *rewriting abstract machine*; a key issue in this regard is compiling conditional equations into unconditional ones [35, 85];
- more experience is also needed in specifying different programming languages as rewrite theories; besides the work in the JavaFAN project, other language specification projects are currently underway at UIUC and at UFF Brazil, including Scheme, ML, OCaml, Haskell, and Pascal;
- more research is also needed on modularity issues; a key question is how to generalize to a true concurrency setting the modular methodology developed in [46]; an important goal would the development of a modular library of rewriting logic definitions of programming language features that could be used to easily define the semantics of a language by putting together different modules in the library.

There is, finally, what we perceive as a promising new direction in teaching programming languages, namely the development of courses and teaching material that use executable rewriting logic specifications as a key way to explain the precise meaning of each programming language feature. This can allow students to experiment with programming language concepts by developing executable formal specifications for them. We have already taught several graduate course at UIUC along these lines with very encouraging results, including a programming language design course and a formal verification course [63, 45].

Acknowledgments. This research has been supported by ONR Grant N00014-02-1-0715 and NSF Grant CCR-0234524. We thank the IJCAR 2004 organizers

for giving us the opportunity of presenting these ideas in an ideal forum. Several of the ideas presented in this work have been developed in joint work with students and colleagues; in particular: (1) the work on Maude is joint work of the first author with all the members of the Maude team at SRI, UIUC, and the Universities of Madrid, Málaga, and Oslo; (2) the work on Java and the JVM is joint work of both authors with Azadeh Farzan and Feng Cheng at UIUC; and (3) ideas on modular rewriting logic definitions have been developed in joint work of the first author with Christiano Braga at UFF Brazil. We thank Feng Chen for help with some of the examples in this paper, to Marcelo D'Amorim for help with editing, and to Mark-Oliver Stehr, Salvador Lucas and Santiago Escobar for their helpful comments on a draft version of this paper.

References

1. H. Baker and C. Hewitt. Laws for communicating parallel processes. In *Proceedings of the 1977 IFIP Congress*, pages 987–992. IFIP Press, 1977.
2. D. Basin and G. Denker. Maude versus Haskell: an experimental comparison in security protocol analysis. In *Proc. 3rd. WRLA*. ENTCS, Elsevier, 2000.
3. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
4. E. Best and R. Devillers. Sequential and concurrent behavior in Petri net theory. *Theoretical Computer Science*, 55:87–136, 1989.
5. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
6. G. Boudol. Computational semantics of term rewriting systems. In *Algebraic Methods in Semantics*, pages 169–236. Cambridge University Press, 1985.
7. 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.
8. C. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Mapping modular SOS to rewriting logic. In *12th International Workshop, LOPSTR 2002, Madrid, Spain*, volume 2664 of *LNCS*, pages 262–277, 2002.
9. C. Braga and J. Meseguer. Modular rewriting semantics in practice. in Proc. *WRLA'04*, ENTCS.
10. G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *ASE'00*, pages 3 – 12, 2000.
11. 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.
12. R. Bruni. *Tile Logic for Synchronized Rewriting of Concurrent Systems.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999. Technical Report TD-1/99. `http://www.di.unipi.it/phd/tesi/tesi_1999/TD-1-99.ps.gz`.
13. R. Bruni and J. Meseguer. Generalized rewrite theories. In *Proceedings of ICALP 2003, 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 252–266, 2003.
14. G. Carabetta, P. Degano, and F. Gadducci. CCS semantics via proved transition systems and rewriting logic. In *Proceedings of WRLA'98, September 1–4, 1998*, volume 15 of *ENTCS*, pages 253–272. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

15. F. Chen, G. Roşu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 197–207, 2003.
16. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.
17. 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.
18. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual. June 2003, `http://maude.cs.uiuc.edu`.
19. 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`.
20. D. Clément, J. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In *Proceedings, France-Japan AI and CS Symposium*, pages 49–89. ICOT, 1986. Also, Information Processing Society of Japan, Technical Memorandum PL-86-6.
21. R. De Nicola and U. Montanari (editors). Selected papers of the 2nd workshop on concurrency and compositionality, March 1990. *Theoretical Computer Science*, 96(1), 1992.
22. P. Degano, F. Gadducci, and C. Priami. A causal semantics for CCS via rewriting logic. *Theoretical Computer Science*, 275(1-2):259–282, 2002.
23. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *Proc. 4th. WRLA*. ENTCS, Elsevier, 2002.
24. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. JavaFAN. `http://fsl.cs.uiuc.edu/javafan`.
25. A. Farzan, F. Cheng, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. To appear in Proc. CAV'04, Springer LNCS, 2004.
26. A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. To appear in Proc. AMAST'04, Springer LNCS, 2004.
27. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
28. F. Gadducci and U. Montanari. The tile model. In G. Plotkin, C. Stirling and M. Tofte, eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, 133–166, 2000.
29. J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.
30. J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In *Formalizing Programming Concepts*, pages 292–309. Springer-Verlag, 1981. LNCS, Volume 107.
31. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal analysis of the remote agent before and after flight. In *the 5th NASA Langley Formal Methods Workshop*, 2000.
32. K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, Aug. 2001. Previous version appeared in Proceedings of the 4th SPIN workshop, 1998.
33. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366 – 381, Apr. 2000.
34. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Willey & Sons, 1990.
35. C. Hintermeier. How to transform canonical decreasing ctrss into equivalent canonical trss. In *4th International CTRS Workshop*, volume 968 of *LNCS*, 1995.

36. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Computing Science Institute, University of Nijmegen, 2000.
37. E. B. Johnsen, O. Owe, and E. W. Axelsen. A runtime environment for concurrent objects with asynchronous method calls. In *Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2004.
38. C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 6:219–249, 1996.
39. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic, 2nd. Edition*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Tech. Report SRI-CSL-93-05, August 1993.
40. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
41. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.
42. J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In *CONCUR'96: Concurrency Theory, 7th International Conference, Pisa, Italy, August 26–29, 1996, Proceedings*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, 1996.
43. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.
44. J. Meseguer. Software specification and verification in rewriting logic. In *Models, Algebras, and Logic of Engineering Software, NATO Advanced Study Institute, July 30 – August 11, 2002*, pages 133–193. IOS Press, 2003.
45. J. Meseguer. Lecture notes on program verification. CS 376, University of Illinois, `http://http://www-courses.cs.uiuc.edu/~cs376/`, Fall 2003.
46. J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. To appear in Proc. AMAST'04, Springer LNCS, 2004.
47. J. Meseguer, K. Futatsugi, and T. Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proceedings of the 1992 International Symposium on New Models for Software Architecture, November 1992*, pages 61–106, 1992.
48. J. Meseguer and U. Montanari. Mapping tile logic into rewriting logic. In *Recent Trends in Algebraic Development Techniques, WADT'97, June 3–7, 1997*, volume 1376 of *LNCS*, pages 62–91. Springer-Verlag, 1998.
49. J. Meseguer and C. L. Talcott. A partial order event model for concurrent objects. In *CONCUR'99, August 24–27, 1999*, volume 1664 of *LNCS*, pages 415–430. Springer-Verlag, 1999.
50. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
51. J. Moore. Inductive assertions and operational semantics. In *Proceedings CHARME 2003*, volume 2860, pages 289–303. Springer LNCS, 2003.
52. 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.
53. J. S. Moore. `http://www.cs.utexas.edu/users/xli/prob/p4/p4.html`.
54. P. D. Mosses. Modular structural operational semantics. Manuscript, September 2003, to appear in *J. Logic and Algebraic Programming*.

55. P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science, Vol. B*. North-Holland, 1990.
56. 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.
57. P. D. Mosses. Pragmatics of modular SOS. In *Proceedings of AMAST'02 Intl. Conf*, pages 21–40. Springer LNCS 2422, 2002.
58. P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. `http://maude.csl.sri.com/papers`.
59. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
60. D. Y. W. Park, U. Stern, J. U. Sakkebaek, and D. L. Dill. Java model checking. In *ASE'01*, pages 253 – 256, 2000.
61. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.
62. J. C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.
63. G. Roşu. Lecture notes on program language design. CS 322, University of Illinois at Urbana-Champaign, Fall 2003.
64. 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.
65. G. Roşu and J. Whittle. Towards certifying domain-specific properties of synthesized code. In *Proceedings, International Conference on Automated Software Engineering (ASE'02)*. IEEE, 2002. Edinburgh, Scotland.
66. 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.
67. L. J. Steggles. Rewriting logic and Elan: Prototyping tools for Petri nets with time. In *Applications and Theory of Petri Nets 2001, 22nd CATPN 2001, June 25–29, 2001*, volume 2075 of *LNCS*, pages 363–381. Springer-Verlag, 2001.
68. M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. In *Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2000.
69. M.-O. Stehr. A rewriting semantics for algebraic nets. In *Petri Nets for System Engineering — A Guide to Modeling, Verification, and Applications*. Springer-Verlag, 2001.
70. M.-O. Stehr. Programming, Specification, and Interactive Theorem Proving — Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory. Doctoral Thesis, Universität Hamburg, Fachbereich Informatik, Germany, 2002. `http://www.sub.uni-hamburg.de/disse/810/`.
71. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework. To appear in Springer LNCS Vol. 2635, 2004.
72. M.-O. Stehr, J. Meseguer, and P. Ölveczky. Rewriting logic as a unifying framework for Petri nets. In *Unifying Petri Nets*, pages 250–303. Springer LNCS 2128, 2001.
73. M.-O. Stehr and C. Talcott. Plan in Maude: Specifying an active network programming language. In *Proc. 4th. WRLA*. ENTCS, Elsevier, 2002.
74. C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000.

75. C. L. Talcott. Interaction semantics for components of distributed systems. In *Proceedings of FMOODS'96*, pages 154–169. Chapman & Hall, 1997.
76. C. L. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285, 2002.
77. P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In *Proc. 4th. WRLA*. ENTCS, Elsevier, 2002.
78. D. Turi. *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University, Amsterdam, 1996.
79. A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.
80. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.
81. A. Verdejo and N. Martí-Oliet. Executing E-LOTOS processes in Maude. In *INT 2000, Extended Abstracts*, pages 49–53, Mar. 2000. Technical report 2000/04, Technische Universitat Berlin.
82. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6. October 10–13, 2000*, volume 183 of *IFIP*, pages 351–366, 2000.
83. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In *Proc. 4th. WRLA*. ENTCS, Elsevier, 2002.
84. P. Viry. Input/output for ELAN. In *Proceedings of WRLA'96, September 3–6, 1996*, volume 4 of *ENTCS*, pages 51–64. Elsevier, Sept. 1996.
85. P. Viry. Elimination of conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999.
86. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.
87. W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. In *Proceedings of Post-CAV Workshop on Advances in Verification*, 2000.
88. M. Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980.