Modular Rewriting Semantics of Programming Languages

José Meseguer[1] and Christiano Braga[2]

[1] University of Illinois at Urbana-Champaign, USA
[2] Universidade Federal Fluminense, Niterói, Brazil

Abstract. We present a general method to achieve modularity of semantic definitions of programming languages specified as rewrite theories. This provides modularity for a language specification method that combines and extends the best features of both SOS and algebraic semantics. The relationship to Mosses' modular operational semantics (MSOS) is explored in detail, yielding a semantics-preserving translation that supports execution and formal analysis of MSOS specifications in Maude.

1 Introduction

This work presents a general method to achieve modularity of semantic definitions of programming languages specified as theories in rewriting logic [15, 5], a logical framework which can naturally represent many different logics, languages, operational semantics formalisms, and models of computation [15, 13, 14]. Since equational logic is a sublogic of rewriting logic, this language specification style generalizes the well-known *algebraic semantics* of programming languages (see, e.g., [12] for an early paper, [23] for the relationship with action semantics, and [11] for a recent textbook). The point of this generalization is that equational logic is well suited for specifying *deterministic* languages, but ill suited for concurrent language specification. In rewriting logic, deterministic features are described by *equations*, but concurrent ones are instead described by *rewrite rules* with a concurrent transition semantics. Our modularity techniques yield also new modularity techniques for algebraic semantics as a special case.

It has also been clear from the early stages [15, 19, 13] that there is a natural semantic mapping of structural operational semantics (SOS) definitions [26] into rewriting logic. In essence, an SOS rule is mapped to a *conditional* rewrite rule. In a sense, we can view rewriting logic semantics as a "conceptual pushout" of algebraic semantics and SOS, that adds a crucial distinction between equations and rules (determinism vs. concurrency) missing in each of those two formalisms. This distiction is of more than academic interest. The point is that, since rewriting with rules R takes place *modulo* the equations E [15], only the rules R contribute to the size of a system's state space, which can be drastically smaller than if all axioms had been given as rules. This, observation, combined with the fact that rewriting logic has several high-performance implementations [1, 10, 7] and associated formal verification tools [8, 14], means that we can use rewriting logic language definitions to obtain practical language analysis tools *for free*. For example, in the JavaFAN formal analysis tool [9], the sematics of the JVM is defined as a rewrite theory in Maude, which is then used to perform formal analyses such as symbolic simulation, search, and LTL model checkig of Java programs with a performance that compares favorably with that of other Java analysis tools. Indeed, the fact that rewriting logic specifications provide in practice an easy way to develop executable formal definitions of programming languages, which can then be subjected to different tool-supported formal analyses, is by now well established [32, 2, 33, 29, 28, 17, 30, 6, 27, 31, 9].

The new question that this work addresses is: how can rewriting logic specifications of programming languages be made *modular*, so that the semantics of each feature can be given once and for all, instead of having to redefine the semantic axioms in a language extension? In this regard, we have learned much from the fact that standard SOS specifications are remarkably *nonmodular* (see [24] and Appendix A) and from Mosses' elegant solution to the SOS modularity problem though his modular structural operational semantics (MSOS) [22, 24, 25, 21]. To maximize modularity and extensibility, the techniques we propose make the semantic rules as abstract and as general as possible using two key ideas: *record inheritance* through associative commutative matching (a technique also used in MSOS); and systematic use of *abstract interfaces*. We compare in detail our modularity techniques with those of MSOS. This takes the form of a translation map τ mapping each MSOS specification to an (also modular) rewrite theory. We prove that τ has very strong semantics-preserving properties, including a bisimilarity result between transition systems for an MSOS specification and its resulting translation. In this regard, this work further advances a line of join work with Hermann Haeusler and Peter Mosses on semantics-preserving translations from MSOS to rewriting logic [3, 2, 4]. The translation τ and its properties are discussed in Section 4. As for the rest of the paper, Section 2 summarizes prerequisites, Section 3 presents our modularity techniques, and Section 5 gives some conclusions.

2 Membership Equational Logic and Rewriting Logic

We gather here prerequisites about membership equational logic (MEL) [16]. and rewriting logic [15, 5]. Maude 2.0 [7] supports all the logical features described below, with a syntax almost identical to the mathematical notation.

2.1 Membership Equational Logic

A MEL *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 $S = \{S_k\}_{k\in K}$

$$\frac{t \in \mathbb{T}_{\Sigma}(X)_k}{(\forall X)\ t \to t}\ \textbf{Reflexivity} \qquad \frac{(\forall X)\ t_1 \to t_2, \qquad (\forall X)\ t_2 \to t_3}{(\forall X)\ t_1 \to t_3}\ \textbf{Transitivity}$$

$$\frac{E \vdash (\forall X)\ t = u, \qquad (\forall X)\ u \to u', \qquad E \vdash (\forall X)\ u' = t'}{(\forall X)\ t \to t'}\ \textbf{Equality}$$

$$\frac{\begin{array}{c} f \in \Sigma_{k_1 \cdots k_n, k}, \qquad t_i, t_i' \in \mathbb{T}_{\Sigma}(X)_{k_i} \ \text{ for } \ i \in \{1, \ldots, n\} \\ t_i' = t_i \ \text{ for } \ i \in \phi(f), \qquad (\forall X)\ t_j \to t_j' \ \text{ for } \ j \in \nu(f) \end{array}}{(\forall X)\ f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')}\ \textbf{Congruence}$$

$$\frac{\begin{array}{c} (\forall X)\ r : t \to t' \text{ if } \bigwedge_{i \in I} p_i = q_i \ \wedge \ \bigwedge_{j \in J} w_j : s_j \ \wedge \ \bigwedge_{l \in L} t_l \to t_l' \in R \\ \theta, \theta' : X \to \mathbb{T}_{\Sigma}(Y), \qquad \theta(x) = \theta'(x) \ \text{ for } \ x \in \phi(t, t') \\ E \vdash (\forall Y)\ \theta(p_i) = \theta(q_i) \ \text{ for } \ i \in I, \qquad E \vdash (\forall Y)\ \theta(w_j) : s_j \ \text{ for } \ j \in J \\ (\forall Y)\ \theta(t_l) \to \theta(t_l') \ \text{ for } \ l \in L, \qquad (\forall Y)\ \theta(x) \to \theta'(x) \ \text{ for } \ x \in \nu(t, t') \end{array}}{(\forall Y)\ \theta(t) \to \theta'(t')}\ \begin{array}{l}\textbf{Nested}\\ \textbf{Replacement}\end{array}$$

Fig. 1. Deduction rules for rewriting logic.

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 $\mathbb{T}_{\Sigma,k}$ and $\mathbb{T}_{\Sigma}(X)_k$ 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 $t, t' \in \mathbb{T}_{\Sigma}(X)_k$ and $s \in S_k$; and Σ*-sentences* are conditional formulae of the form $(\forall X)\ \varphi$ if $\bigwedge_i p_i = q_i \ \wedge \ \bigwedge_j w_j : s_j$, where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i, and w_j are in X. A MEL theory is a pair (Σ, E) with Σ a MEL signature and E a set of Σ-sentences. We refer to [16] for the detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, and initial and free algebras. 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 and Deduction

We present the general version of rewrite theories over **MEL** theories defined in [5]. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, \phi, R)$ consisting of: (i) a **MEL** theory (Σ, E); (ii) a function $\phi\colon \Sigma \to \wp_f(\mathbb{N})$ assigning to each function symbol $f\colon k_1 \cdots k_n \to k$ in Σ a set $\phi(f) \subseteq \{1, \ldots, n\}$ of *frozen argument positions*; (iii) a set R of (universally quantified) labeled conditional rewrite rules r having the general form

$$(\forall X)\ r : t \to t' \text{ if } \bigwedge_{i \in I} p_i = q_i\ \wedge\ \bigwedge_{j \in J} w_j : s_j\ \wedge\ \bigwedge_{l \in L} t_l \to t'_l \qquad (1)$$

where, for appropriate kinds k and k_l in K, $t, t' \in \mathbb{T}_\Sigma(X)_k$ and $t_l, t'_l \in \mathbb{T}_\Sigma(X)_{k_l}$ for $l \in L$.

The function ϕ specifies which arguments of a function symbol f *cannot be rewritten*, which are called *frozen positions*. Note that if the ith position of f is frozen, then in $f(t_1, \ldots, t_n)$ any subterm of t_i becomes also frozen. That is, the freezing idea extends to subterms and in particular to variables. Given two terms t, t' we can then define the sets $\phi(t, t')$ and $\nu(t, t')$ of their frozen (resp. unfrozen) variables (see [5]). Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, a *sequent* of $\mathcal{R}$ is a pair of (universally quantified) terms of the same kind t, t', denoted $(\forall X) t \to t'$ with $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ a set of kinded variables and $t, t' \in \mathbb{T}_\Sigma(X)_k$ for some k. We say that $\mathcal{R}$ *entails* the sequent $(\forall X)\ t \to t'$, and write $\mathcal{R} \vdash (\forall X)\ t \to t'$, if the sequent $(\forall X)\ t \to t'$ can be obtained by means of the inference rules in Figure 1. **(Reflexivity)**, **(Transitivity)**, and **(Equality)** are the usual rules for idle rewrites, concatenation of rewrites, and rewriting modulo the **MEL** theory E. **(Congruence)** allows rewriting the arguments of a generalized operator, subject to the condition that frozen arguments must stay idle (note that $t'_i = t_i$ is syntactic equality). Any unfrozen argument can still be *concurrently rewritten*, as expressed by the premise $(\forall X)\ t_j \to t'_j$ for $j \in \nu(f)$. **(Nested Replacement)** characterizes the concurrent application of a rewrite rule in its most general form (1). It specifies that for any rewrite rule $r \in R$ and for any (kind-preserving) substitution θ such that the condition of r is satisfied when θ is applied to all terms p_i, q_i, w_j, t_l, t'_l involved, then it is possible to apply the rewrite r to $\theta(t)$. Moreover, if θ' is a second (kind-preserving) substitution for the variables in X such that θ and θ' coincide on all frozen variables $x \in \phi(t, t')$ (second line of premises), while the rewrites $(\forall Y)\ \theta(x) \to \theta'(x)$ are provable for the unfrozen variables $x \in \nu(t, t')$ (last premise), then such nested rewrites can be applied *concurrently* with r.

3 Modular Language Specifications in Rewriting Logic

Modularity is only meaningful in the context of an *incremental* specification, where syntax and corresponding semantic axioms are introduced for groups of related features. We can describe an *incremental* presentation of the syntax of a programming language $\mathcal{L}$ as an indexed family of syntax definitions $\{\mathcal{L}_i\}_{i\in I}$, where the index set I is a *poset* with a top element $\top$, such that: (i) if $i \leq j$, then $\mathcal{L}_i \subseteq \mathcal{L}_j$, and (ii) $\mathcal{L}_\top = \mathcal{L}$. An *incremental rewriting semantics* for $\mathcal{L}$ is then an indexed family of rewrite theories $\{\mathcal{R}_{\mathcal{L}_i}\}_{i\in I}$, with $\mathcal{R}_{\mathcal{L}_i}$ defining the semantics of the language fragment $\mathcal{L}_i$. Modularity of the incremental rewriting semantics $\{\mathcal{R}_{\mathcal{L}_i} = (\Sigma_i, E_i, \phi_i, R_i)\}_{i\in I}$ means in essence two things. First of all, it should satisfy the following *monotonicity property*: if $i \leq j$, then there is a theory inclusion $\mathcal{R}_{\mathcal{L}_i} \subseteq \mathcal{R}_{\mathcal{L}_j}$. This is not easy to achieve; for example, Appendix A presents typical SOS rules for a simple language that are not monotonic.

However, one can always achieve monotonicity *a posteriory*, by carving out submodules of the top module $\mathcal{R}_{\mathcal{L}_\top}$ for each of the language fragments $\mathcal{L}_i$. In the example in Appendix A, this would correspond to claiming that the rules in the last language extension giving semantics to the earlier features are their specification. This would of course be cheating, as can be immediately recognized by asking the specifier to tell us what the SOS rules are going to be when new features, for example concurrent synchronization, are added in a further extension. Therefore, besides monotonicity, we need the second requirement of *extensibility*. This, together with monotonicity, means that the semantics of each language feature can be defined *once and for all*, so that we never have to *retract* earlier semantic definitions in a later language extension. One is then interested in *methods* that can be used to develop incremental rewriting semantics definitions of programming languages that are as modular as possible, in both the monotonic and extensible senses.

The method we propose uses pairs, called *configurations*; the first component is the *program text*, and the second a *record* with the different *semantic entities* that change as the program is computed. That is, we organize all the semantic entities associated to a program's computation in a record data structure. For example, one of the record's fields may be the *store*, another the *environment* of declarations, and yet another the *traces* left by a concurrent process' execution. We can specify configurations in Maude with the following membership equational theory (a functional module importing the `RECORD` module shown later in `protecting` mode, that is, adding no more data ("no junk") and no new equalities ("no confusion") to records; the `ctor` keyword indicates a *constructor*):

```
fmod CONF is
 protecting RECORD .
 sorts Program Conf .
 op <_,_> : Program Record -> Conf [ctor] .
endfm
```

The first key modularity technique is *record inheritance*, which is accomplished through pattern matching *modulo* associativity, commutativity, and identity. Features added later to a language may necessitate adding new semantic components to the record; but the axioms of older features can be given once and for all in full generality: they will apply just the same with new components in the record. Here is the Maude specification of the membership equational theory of records. Note Maude's convention of identifying kinds with connected components in the subsort inclusion poset, and naming them as equivalence classes of sorts in such components. For example, `[PreRecord]` denotes the kind determined by the connected component {`Field`,`PreRecord`}.

```
fmod RECORD is
 sorts Index Component Field PreRecord Record Truth .
 subsort Field < PreRecord .
 op tt : -> Truth
 op null : -> PreRecord [ctor] .
 op _,_ : PreRecord PreRecord -> PreRecord [ctor assoc comm id: null] .
 op _:_ : [Index] [Component] -> [Field] [ctor] .
 op {_} : [PreRecord] -> [Record] [ctor] .
 op duplicated : [PreRecord] -> [Truth] .
 var I : Index . vars C C' : Component . var PR : PreRecord .
 eq duplicated((I : C),(I : C'), PR) = tt .
 cmb {PR} : Record if duplicated(PR) =/= tt .
endfm
```

A `Field` is defined as a pair of an `Index` and a `Component`; illegal pairs will be of kind `[Field]`. A `PreRecord` is a possibly empty (`null`) multiset of fields, formed with the union operator `_,_` which is declared to be *associative* (`assoc`), *commutative* (`comm`) and to have `null` as its *identity* (`id`). Maude will then apply all equations and rules *modulo* such equational axioms [7]. Note the conditional membership (`cmb`) defining a `Record` as an "encapsulated" `PreRecord` with no duplicated fields. *Record inheritance* means that we can always consider a record with more fields as a special case of one with fewer fields. For example, a record with an environment component indexed by `env` and a store component indexed by `st` can be viewed as a special case of a record with just the environment component. Matching modulo associativity, commutativity, and identity supports record inheritance, because we can always use an extra variable `PR` of sort `PreRecord` to match *any extra fields the record may have.* For example, a function `get-env` extracting the environment component can be defined by

```
 eq get-env({env : E , PR}) = E .
```

and will apply to records with any extra fields that are matched by `PR`.

The second key modularity technique is the systematic use of *abstract interfaces.* That is, the sorts specifying key syntactic and semantic entities are *abstract sorts* for which we:

- only specify the *abstract functions* manipulating them, that is, a given *signature*, or *interface*, of abstract sorts and functions; *no axioms* are specified about such functions *at the level of abstract sorts*;

– in a language specification no *concrete* syntactic or semantic sorts are ever identified with abstract sorts: they are always either specified as *subsorts* of corresponding abstract sorts, or mapped to abstract sorts by *coercions*; it is *only at the level of such concrete sorts* that *axioms* about abstract or auxiliary functions are specified.

This means that we *make no apriori ontological commitments* as to the nature of the syntactic or semantic entities. It also means that, since the only commitments ever made happen always at the level of *concrete sorts*, one remains forever free to introduce new meaning and structure in any language extension.

A third modularity technique regards the form of the rules. We require that the only new rewrite rules in the i^{th} rewrite theory $\mathcal{R}_{\mathcal{L}_i}$ are semantic rules

$$\langle f(t_1, \ldots, t_n), u\rangle \longrightarrow \langle t', u'\rangle \text{ if } C,$$

where f is a new language feature, e.g., `if-then-else`, introduced in $\mathcal{L}_i$, u and u' are record expressions and u contains a variable `PR` of sort `PreRecord` standing for unspecified additional fields and allowing the rule to match by record inheritance. In addition, the following *information hiding* discipline should be followed in u, u', and in any record expressions appearing in C: besides basic record syntax, only function symbols appearing in the *abstract interfaces* of some of the record's fields can appear in record expressions; any auxiliary functions defined in concrete sorts of those field's components should never be mentioned. This information hiding makes the rules highly extensible, because the concrete representations of the auxiliary semantic entities can be changed or extended without having to change the rules at all. For example, we can change or extend the internal representations of traces, stores, or environments, without the rules being affected in any way: all we have to do is add new equations defining the semantics of the abstract functions on the new concrete data representations. For two interesting applications of this modularity discipline, one in which the semantics of CCS is extended from the strong transition semantics to the weak transition semantics, and another in which the bc sublanguage of C is enriched with annotations for physical units in the style of [6], so that the values stored now need to be pairs of a number and a unit expression, see [18] and `http://formal.cs.uiuc.edu/meseguer/modular`. Another example illustrating our general methodology is given in Appendix B, which gives a modular rewriting semantics specification of the language in Appendix A.

The combination of these three techniques can be of great help in making semantic definitions modular and easily extensible. That is, we can develop in this way modular incremental semantic definitions for a language $\mathcal{L}$ as a poset-indexed hierarchy $\{\mathcal{R}_{\mathcal{L}_i} = (\Sigma_i, E_i, \phi_i, R_i)\}_{i\in I}$ of rewrite theory *inclusions*, with the full language definition as the top theory in the hierarchy and with the theory `CONF` —which contains `RECORD`— as the bottom of the hierarchy. By following the methods described above, such a modular definition will then be much more easily extensible than if such methods had not been followed. As we explain in more detail in Section 4, the above methods are closely related to Mosses' MSOS methodology; however, besides the fact that we are exploring such methods in a different semantic framework, it appears that our systematic use of abstract interfaces is an aspect considerably less developed in the MSOS approach.

An important variant of our approach is to choose the MEL sublogic of rewriting logic as the *logical framework* in which to define the semantics of a language. As argued in [17], this is a perfectly good possibility for *sequential* or *deterministic* languages, but is typically a bad choice for concurrent languages. Of course, in this case the semantics is no longer given by rewrite rules, but by *conditional equations* of the form,

$$\langle f(t_1, \ldots, t_n), u \rangle = \langle t', u' \rangle \ \text{ if } \ C.$$

This variant provides modularity techniques for a long strand of work in the so-called *algebraic* or *initial algebra semantics* of programming languages (see, e.g., [12, 23, 11]). In fact, as explained in the Introduction, the best approach in practice is to *combine* the equational and the rewriting variants of the modular language specification methodology just described. This is most natural in a rewriting logic context, because of its explicit distinction between equations and rules.

3.1 Controlling Rewrite Steps in Conditions

When relating SOS rules to rewrite rules a few technicalities are needed to obtain an exact correspondence. The key issue is the *number of steps* of rewrites in conditions. In an SOS rule

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

the rewrites $P_i \longrightarrow P_i'$ in the condition are *one-step rewrites*. By contrast, in a rewrite rule

$$Q \longrightarrow Q' \ \text{ if } \ P_1 \longrightarrow P_1' \ \wedge \ \ldots \ \wedge \ P_n \longrightarrow P_n',$$

because of the **(Reflexivity)** and **(Transitivity)** inference rules of rewriting logic (see Figure 1) the rewrites $P_i \longrightarrow P_i'$ in the condition are considerably more general: they can have zero, one, or more steps of rewriting. The point is that, by definition, in rewriting logic *all finitary computations are always derivable as sequents*, whereas in SOS they are not, unless special provisions are made such as a big-step style or adding transitivity as an explicit SOS rule. As a consequence, SOS rules may not be directly usable as an interpreter. The question we now address is how to accomplish two simultaneous goals: (i) to represent SOS rules in an exact way, with one step rewrites in conditions; and (ii) to get also all finitary computations in the rewriting logic representation, so that we always get a language interpreter, even when the SOS rules do not directly provide one. We present a method that will work for any of the rewrite theories characterized in this Section and will accomplish goals (i) and (ii): First of all, we extend the module `CONF` to a system module (rewrite theory):

```
mod RCONF is extending CONF .
 op {_,_} : [Program] [Record] -> [Conf] [ctor] .
 op [_,_] : [Program] [Record] -> [Conf] [ctor] .
 vars P P' : Program .  vars R R' : Record .
 crl [step] : < P , R > => < P' , R' > if { P , R } => [ P' , R' ] .
endm
```

We furthermore require semantic rewrite rules to be of the form,

$$\{t, u\} \longrightarrow [t', u'] \text{ if } \{v_1, w_1\} \longrightarrow [v'_1, w'_1] \wedge \ldots \wedge \{v_n, w_n\} \longrightarrow [v'_n, w'_n] \wedge C, \tag{2}$$

where $n \geq 0$, and C is a (possibly empty) equational condition involving only equations and memberships. Note that a rewrite theory $\mathcal{R}$ containing only `RCONF` and rules of the form (2) will be such that any proof of a rewrite

$$\mathcal{R} \vdash \langle v, w \rangle \longrightarrow \langle v', w' \rangle$$

can be expressed (up to the equational equivalence of proofs defined in [5]) as either an application of the **(Equality)** and **(Reflexivity)** inference rules, or as repeated applications (no application if $n = 1$) of the **(Transitivity)** rule to proofs of rewrites of the form,

$$\langle v, w \rangle = \langle v_0, w_0 \rangle \longrightarrow \langle v_1, w_1 \rangle \longrightarrow \ldots \langle v_{n-1}, w_{n-1} \rangle \longrightarrow \langle v_n, w_n \rangle = \langle v', w' \rangle, \tag{3}$$

where each rewrite in the sequence is obtained by application of the **(Nested Replacement)** inference rule to the `step` rule, and by **(Equality)**. Any such application of the `step` rule exactly mimics a one-step rewrite with a rule of the form (2) in its condition, so the sequences (3) are the finitary *computations*.

4 Relationship to MSOS

Our modular rewriting logic ideas have a close relationship to a new formulation of MSOS initiated in [25] and further developed by Peter Mosses under the name of *definitive semantics* [20, 21]. This allows us to define a quite succinct mapping from MSOS to rewriting logic that is semantics-preserving in a very strong sense, and is modularity-preserving; that is, MSOS specifications are mapped to modular rewriting logic specifications in the sense of Section 3. In Mosses' definitive semantics, labeled transitions are of the form $t \xrightarrow{u} t'$, where t, t' are *program expressions* (which can involve values), and u is a *record* which specifies the semantic information *before and after* the transition takes place. This is accomplished by postulating that the indices of the record are classified in three different types:

- *read-only*, where the index, say i, is a name with no additional structure;
- *write-only*, where the index has primed form, that is, is of the form i'; and
- *read-write*, where there are in fact *two* indices in the record: a plain i, and its primed variant i'.

Furthermore, the values of any write only index must range over some *free monoid of lists*, with an associative append operation $\_.\_$ and neutral element nil. For example, an environment index *env* will be read-only, a trace index *tr'* for a concurrent process will be write only, and a store index *st* will be read-write. As usual, the primed indices indicate the relevant changes *after* the transition takes place. Such records can be naturally viewed as the arrows of a *label category*, so that several-step *computations*[3] can be defined by composing the labels and forgetting the intermediate stages as follows: $(t \xrightarrow{u} t'); (t' \xrightarrow{u'} t'') = (t \xrightarrow{u;u'} t'')$. For the composition $u;u'$ to be defined, we must have (in usual record notation) $u.i = u'.i$ for each read-only index i, and $u.j' = u'.j$ for each read-write index j. The composition $u;u'$ then has $(u;u').i = u.i = u'.i$, $(u;u').j = u.j$, $(u;u').j' = u'.j'$, and $(u;u').k = (u.k).(u'.k)$ for each write-only index k. *Identities* are then records u such that $u.j = u.j'$, and $u.k = nil$. Note that we can easily axiomatize these records and their composition and units in an extension of our `RECORD` module. In particular, we will have a subsort `IRecord < Record` of identity records, another subsort `IPreRecord < PreRecord` of identity prerecords, and a partial record composition operator $\_;\_$. In what follows we will use variables $X, X' \ldots$, of sort `Record`, variables $U, U', \ldots$, of sort `IRecord`, variables $PR, PR' \ldots$, of sort `PreRecord`, and variables $UPR, UPR', \ldots$, of sort `IPreRecord`.

In MSOS a rule defining the semantics of a language feature f has the general form,

$$\frac{v_1 \xrightarrow{u_1} v_1' \ \ldots \ v_n \xrightarrow{u_n} v_n' \quad cnd}{f(t_1, \ldots, t_n) \xrightarrow{u} t'} \tag{4}$$

where $f(t_1, \ldots, t_n), t'$, and the v_i, v_i' are *program expressions* (which can involve values), $u, u_1, \ldots, u_n$ are record expressions, and *cnd* is a side condition involving equations and perhaps predicates. The key idea is that matching of such MSOS rules uses *record inheritance*. Our concrete notation is in essence the notational variant of MSOS contemplated in Footnote 2 of [21]. For example, rule `[10]` in Appendix B could be expressed in our notational variant of MSOS as follows:

$$\frac{\sigma' = \sigma[l \mapsto v]}{l := v \xrightarrow{\{st:\sigma, st':\sigma', UPR\}} noop} \tag{5}$$

A definitive MSOS specification $\mathcal{S}$ has rules of the form (4), but must also specify: (i) the *syntax* of programs in the language $\mathcal{L}$ of interest, and (ii) the

[3] A somewhat subtle point explained in what follows is that to organize the computations themselves as a category we need some more structure on the objects.

semantic entities used in the different record components. We choose membership equational logic to specify (i) and (ii). Therefore, in what follows we assume that an MSOS specification is in essence equivalent to a triple (Σ, E, R), with (Σ, E) a membership equational theory containing abstract sorts `Program` and `Component`, and importing the above-sketched extension of the `RECORD` specification as a subtheory, and with no other operations in Σ having kinds in the `RECORD` extension as arguments, except for the `[Component]` kind. The rules R are then MSOS rules of the general form (4), where $f(t_1, \ldots, t_n), t'$, and the v_i, v'_i are terms of sort `Program`, $u, u_1, \ldots, u_n$ are expressions of sort `Record`, and *cnd* is a conjunction of equations and memberships. Furthermore, we assume that all MSOS rules in R are in the following *normal form*: (i) the side condition *cnd* does not involve any record, field, index, or `[Truth]` (different from `[Bool]`) expressions in its terms or subterms; and (ii) a record expression appearing in either the premisses or the conclusion is either: (1) a variable of the general form X, or U; or (2) a constructor term of the general form $\{i_1 : w_1, \ldots, i_n : w_n, PR\}$, or $\{i_1 : w_1, \ldots, i_n : w_n, UPR\}$, with $n \geq 0$, some of the indices perhaps primed, and the w_j terms with sorts in the corresponding field components. For example, rule `[9]` in in Appendix B might be expressed as the MSOS rule

$$\frac{U = \{env : \rho, UPR\} \qquad v = \rho(x)}{x \xrightarrow{U} v}$$

which fails to satisfy (i) above, but we have an equivalent normal form

$$\frac{v = \rho(x)}{x \xrightarrow{\{env:\rho, UPR\}} v} \tag{6}$$

The desired translation is a mapping $\tau : (\Sigma, E, R) \mapsto (\Sigma', E', \phi, R')$ where:

1. (Σ', E') is obtained from (Σ, E) by: (i) omitting all the primed indices and their related equations and memberships from the record subspecification (but adding the unprimed version of each write-only index); (ii) defining subsorts `ROPreRecord`, `RWPreRecord`, and `WOPreRecord` (all containing the constant `null`) of the `PreRecord` sort, corresponding to those parts of the record involving read-only, read-write, and write-only fields (we use variables $A, A', \ldots$, $B, B', \ldots$, and $C, C', \ldots$, to range over those respective subsorts); (iii) In `WOPreRecord` we also equationally axiomatize a *prefix predicate* $\sqsubseteq$, where $C \sqsubseteq C'$ means that for each write-only field k the string $C.k$ is a (possibly identical) prefix of the string $C'.k$; and (iv) adding the signature of the module `RCONF`;
2. ϕ declares all operators in Σ' as unfrozen; and
3. R' contains the `step` rule in `RCONF`, as well as for each MSOS rule in R, in the normal form described above, a corresponding rewrite rule of the form,

$$\{f(t_1, \ldots, t_n), u^{pre}\} \longrightarrow [t', u^{post}]$$
$$\text{if } \{v_1, u_1^{pre}\} \longrightarrow [v'_1, u_1^{post}] \ \wedge \ \ldots \ \wedge \ \{v_n, u_n^{pre}\} \longrightarrow [v'_n, u_n^{post}] \ \wedge \ cnd',$$

where for each record expression u in the MSOS rule, u^{pre} and u^{post} are defined as follows. For u a record expression of the general form X or $\{PR\}$, u^{pre} is a record expression of the form $\{A, B, C\}$, and u^{post} has the form $\{A, B', C\}$. For u a record expression of the general form U or $\{UPR\}$, u^{pre} is of the general form R or $\{PR\}$, and $u^{pre} = u^{post}$. Otherwise, if u is of the general form $\{i_1 : w_1, \ldots, i_n : w_n, PR\}$, with $n \geq 1$. Then u^{pre} and u^{post} are record expressions similar to u where: (i) if a read-only field expression $i : w$ appears in u, then it appears in both u^{pre} and u^{post}; (ii) if a write-only field expression $i' : w$ appears in u, if u labels the conclusion, then u^{pre} contains a field expression of the form $i : l$, with l a new list variable in the corresponding data type, and u^{post} contains a field expression of the form $i : l.w$ (if u labels a condition, u^{pre} contains $i : nil$, and u^{post} contains $i : w$); (iii) if a read-write pair of field expressions $i : w, i' : w'$ appear in u, then u^{pre} contains $i : w$, and u^{post} contains $i : w'$; and (iv) PR is translated in u^{pre} as A, B, C, and in u^{post} as A, B', C'. Finally, if u is of the general form $\{i_1 : w_1, \ldots, i_n : w_n, UPR\}$, with $n \geq 1$. Then u^{pre} and u^{post} are record expressions like u where cases (i)–(iii) as handled as before, and (iv) UPR is translated in both u^{pre} and u^{post} as PR. Furthermore, the condition cnd' is either cnd itself, or is obtained by conjoining to cnd the prefix predicate $C \sqsubseteq C'$ in case subexpressions of the form A, B, C, and A, B', C' were introduced in the terms u^{pre} and u^{post} in the conclusion. We assume throughtout some reasonable naming conventions, such as using different translated names for different variables, introducing the same new names for repeated variables, avoiding variable capture, and so on.

For example, the MSOS rule (5) would be translated as the conditional rewrite rule:

$$\{l := v, \{(st : \sigma), PR\}\} \longrightarrow [noop, \{(st : \sigma'), PR\}] \quad \text{if} \quad \sigma' = \sigma[l \mapsto v].$$

The translation τ just defined is *semantics-preserving* in a very strong sense. To make this clear, we need to discuss the *semantic models* associated to (finite) *computations* in both formalisms. We shall focus on transitions involving *ground* terms t, t' of sort `Program`, and with u a ground record expression. First of all, note that an MSOS specification $\mathcal{S}$ defines a *category* $\mathbb{C}_{\mathcal{S}}$ of *finite computations*, whose arrows have the form $\langle t, u\rangle \xrightarrow{w} \langle t', u'\rangle$, where $u = dom(w)$, and $u' = cod(w)$ are the source and target identity records of the label w, and for some $n \geq 0$ there are n composable transitions (i.e., their labels are composable) derivable from $\mathcal{S}$,

$$t \xrightarrow{w_1} t_1, \; \ldots \; t_{n-1} \xrightarrow{w_n} t'$$

such that $w = w_1; \ldots; w_n$. Note that the objects of $\mathbb{C}_{\mathcal{S}}$ must be pairs $\langle t, u\rangle$, wose identity arrow is u, since without the second component the identities of $\mathbb{C}_{\mathcal{S}}$ would be underdetermined. By consideing the labeled subgraph of $\mathbb{C}_{\mathcal{S}}$ determined by those computations corresponding to one-step transitions we obtain *labeled transition system*[4] that we denote $\mathbb{L}_{\mathcal{S}}$.

[4] Note that this is a more detailed labeled transition system than the one associated to $\mathcal{S}$ in [21], since the states are pairs $\langle t, u\rangle$.

In rewriting logic, the simplest model associated to a rewrite theory $\mathcal{R}$ is its *initial reachability model* $\mathbb{T}_{Reach(\mathcal{R})}$ [5], which defines the $\mathcal{R}$-reachability relation $[t] \longrightarrow [t']$ between equivalence classes[5] of ground terms modulo the equations in $\mathcal{R}$ by the equivalence, $[t] \longrightarrow [t'] \quad \Leftrightarrow \mathcal{R} \vdash t \longrightarrow t'$. In particular, for $\mathcal{R} = \tau(\mathcal{S})$, we can restrict this model to the sort `Conf` and, because of the **(Transitivity)** and **(Reflexivity)** inference rules, we then get a *preorder relation* on equivalence classes of `Conf` ground terms modulo the equations E' in $\tau(\mathcal{S})$, that is, a preorder *category* $\mathbb{T}_{Reach(\tau(\mathcal{S}))}|_{\texttt{Conf}}$. As pointed out in Section 3.1, we will have a $\tau(\mathcal{S})$-reachability relation $\langle v, w\rangle \longrightarrow \langle v', w'\rangle$ iff for some $n \geq 0$, there is a sequence of the form (3) where each step is obtained by application of the **(Nested Replacement)** inference rule to the `step` rule, and by **(Equality)**. Any such application of the `step` rule exactly mimics a one-step rewrite with a rule $r' \in R'$, which is the translation of an MSOS rule $r \in R$ in $\mathcal{S}$. The semantics-preserving nature of τ takes the form of a *functor*

$$\pi : \mathbb{T}_{Reach(\tau(\mathcal{S}))}|_{\texttt{Conf}} \longrightarrow \mathbb{C}_{\mathcal{S}}$$

surjective on objects and arrows and defined on arrows by:

$$\pi : (\langle t, w\rangle \longrightarrow \langle t', w'\rangle) \mapsto (\langle t, \rho(w)\rangle \stackrel{w \mapsto w'}{\longrightarrow} \langle t', \rho(w')\rangle)$$

where the function ρ maps each record w without primed indices in (Σ', E') to an identity record $\rho(w)$ in (Σ, E) by leaving all read-only fields untouched, adding a primed copy of each read-write index (with same value), and making all write only fields primed and all with value nil; and where the label $w \mapsto w'$ is defined as follows. For a read-only index i, w, w', and $w \mapsto w'$ all agree on the field $i : x$; for a write-only index i, if w contains the field $i : l$, then w' will contain a field of the form $i : l.l'$, and $w \mapsto w'$ contains the field $i' : l'$; for a read-write index i, if w contains the field $i : x$, and w' the field $i : y$, then $w \mapsto w'$ contains the fields $i : x$ and $i' : y$. The well-definedness of this functor follows by induction on the length of the rewrites/computations from the strong bisimulation result for one-step rewrites proved below. Furthermore, it also requires showing the well-definedness of the operation $w \mapsto w'$. This follows easily by induction on the lenght of the rewrites from the following lemma, which is itself an easy consequence of the definition of u^{pre} and u^{post} in the translation τ:

Lemma 1. *In any one-step rewite $\langle t, w\rangle \longrightarrow \langle t', w'\rangle$, the record w' has the same read-only fields as w, and the value of any write-only field in w is a prefix of the corresponding value in w'.*

We can associate to $\tau(\mathcal{S})$ the labeled transition system $\mathbb{L}^{\pi}_{\mathbb{T}_{Reach(\tau(\mathcal{S}))}}$, whose transitions are of the form $\langle t, w\rangle \stackrel{w \mapsto w'}{\longrightarrow} \langle t', w'\rangle$, where $\langle t, w\rangle \longrightarrow \langle t', w'\rangle$ is a *one-step* $\tau(\mathcal{S})$-rewrite. The semantics-preserving nature of τ can be further expressed by the following theorem (a proof sketch is given in Appendix C):

[5] To keep the notation lighter, in what follows we will often leave implicit the equivalence class notation, and will denote reachability using representative terms.

Theorem 1. *(Strong Bisimulation). The projection function* $\pi : \langle t, w\rangle \mapsto \langle t, \rho(w)\rangle$, *together with its inverse relation* π^{-1}, *define a* strong bisimulation *of labeled transition systems* $\pi : \mathbb{L}^{\pi}_{\mathbb{T}_{Reach(\tau(\mathcal{S}))}} \longrightarrow \mathbb{L}_{\mathcal{S}}$.

5 Concluding Remarks

We have presented a general method to make semantic definitions of programming languages both modular and executable in rewriting logic. It would be natural to extend these techniques in the direction of "true concurrency." The point is that SOS provides an *interleaving semantics*, based on labeled transition systems, whereas rewriting logic provides a "true concurrency" semantics, in which many rewrites can happen concurrently. This is one of the reasons for a gradual shift towards so-called *reduction semantics* for concurrent languages, which is just another name for semantics defined by rewrite theories. Although an interleaving semantics remains a possible choice, the SOS idea of executing a *single program* becomes less natural when languages are concurrent or even mobile. Therefore, an interesting question to explore is how language specifications based on reduction semantics can be made modular.

We have also discussed the relationship to MSOS. Our new translation from MSOS to rewriting logic makes available in a sound and simple way the possibility of executing MSOS specifications in Maude. As already done for a previous translation [2], this can be the basis of a tool to execute MSOS specifications. Since Maude has breadth-first search, an efficient LTL model checker [7], and several other formal tools [8, 14], MSOS specifications can then be formally analyzed in various ways. Besides the example in Appendix B, we have developed several variants of a modular rewriting semantics for CCS and for the bc language (available in `http://formal.cs.uiuc.edu/meseguer/modular`). More experimentation to validate and further refine our methods remains ahead.

Acknowledgments. This research has been supported by ONR Grant N00014-02-1-0715, NSF Grant CCR-0234524, and by CNPq under processes 552192/2002-3 and 300294/2003-4. We have benefitted much from our collaboration with Hermann Haeusler and Peter Mosses on the MSOS-rewriting logic connection; and from Peter Mosses' very helpful comments that have helped us formulate a precise comparison between his work and ours. We also thank Narciso Martí-Oliet and Grigore Roşu for their detailed comments on an earlier draft.

References

1. 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.
2. 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.

3. C. Braga, E. H. Haeusler, J. Meseguer, and P. Mosses. Maude Action Tool: Using reflection to map action semantics to rewriting logic. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Springer LNCS*, pages 407–421, 2000.
4. C. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Mapping modular SOS to rewriting logic. In M. Leuschel, editor, *12th International Workshop, LOPSTR 2002, Madrid, Spain*, volume 2664 of *Springer LNCS*, pages 262–277, 2002.
5. R. Bruni and J. Meseguer. Generalized rewrite theories. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proceedings of ICALP 2003, 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Springer LNCS*, pages 252–266, 2003.
6. F. Chen, G. Roşu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Springer LNCS*, pages 197–207, 2003.
7. 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`.
8. 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`.
9. A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. Manuscript, Jan. 2004, CS Dept., Univ. of Illinois at Urbana-Champaign, submitted for publication.
10. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
11. J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.
12. J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, pages 292–309. Springer-Verlag, 1981. LNCS, Volume 107.
13. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *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.
14. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
15. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
16. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.
17. J. Meseguer. Software specification and verification in rewriting logic. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software, NATO Advanced Study Institute, Marktoberdorf, Germany, July 30 – August 11, 2002*, pages 133–193. IOS Press, 2003.
18. J. Meseguer and C. Braga. Modular rewriting semantics in practice. To appear in Proc. *WRLA'04*, ENTCS.
19. 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, Tokyo, Japan, November 1992, pages 61–106. Research Institute of Software Engineering, 1992.
20. P. D. Mosses. Definitive semantics. Version 0.2, May 31, 2003, `http://www.mimuw.edu.pl/~mosses/DS-03`.
21. P. D. Mosses. Modular structural operational semantics. Manuscript, September 2003, to appear in *J. Logic and Algebraic Programming*.
22. P. D. Mosses. Semantics, modularity, and rewriting logic. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, Vol. 15, North Holland, 1998.
23. P. D. Mosses. Unified algebras and action semantics. In *Proc. Symp. on Theoretical Aspects of Computer Science, STACS'89*. Springer LNCS 349, 1989.
24. 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.
25. P. D. Mosses. Pragmatics of modular SOS. In *Proceedings of AMAST'02, 9th Intl. Conf. on Algebraic Methodology and Software Technology*, pages 21–40. Springer LNCS 2422, 2002.
26. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.
27. 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.
28. M.-O. Stehr and C. Talcott. Plan in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.
29. P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.
30. A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.
31. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.
32. A. Verdejo and N. Martí-Oliet. Executing and verifying CCS in Maude. Technical Report 99-00, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid; also, `http://maude.cs.uiuc.edu`.
33. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

A SOS Specifications are not Modular

To illustrate the SOS modularity problem, let us consider the following SOS rules (adapted from [3]) for the gradual evaluation of expressions e to their values v. An expression is an addition of expressions or a natural number value.

$$\begin{array}{ll} e ::= v \mid e_1 \texttt{ + } e_2 & \\ v ::= n & ,\ n \in \mathbb{N} \end{array}$$

$$\frac{n = n_1 + n_2}{n_1 \texttt{ + } n_2 \to n} \qquad \frac{e_1 \to e_1'}{e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{e_2 \to e_2'}{v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'}$$

In a functional extension the above rules have to be redefined, because they now must involve both expressions and an environment.

$$\begin{array}{l} e ::= \ldots \mid x \mid e_1\ e_2 \mid \lambda x(e) \\ v ::= \ldots \mid \mathit{closure}(e, \rho) \\ x \in Var,\ \rho \in \mathit{Env} = \mathit{Var} \to \mathit{Val} \end{array}$$

$$\frac{n = n_1 + n_2}{\rho\ \vdash\ n_1 \texttt{ + } n_2 \to n}$$

$$\frac{\rho\ \vdash\ e_1 \to e_1'}{\rho\ \vdash\ e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{\rho\ \vdash\ e_2 \to e_2'}{\rho\ \vdash\ v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'}$$

$$\frac{\rho\ \vdash\ e_1 \to e_1'}{\rho\ \vdash\ e_1\ e_2 \to e_1'\ e_2} \qquad \frac{\rho\ \vdash\ e_2 \to e_2'}{\rho\ \vdash\ v_1\ e_2 \to v_1\ e_2'}$$

$$\frac{}{\rho\ \vdash\ \lambda x(e) \to \mathit{closure}(\lambda x(e), \rho)}$$

$$\frac{\rho'[x \mapsto v]\ \vdash\ e \to e'}{\rho\ \vdash\ \mathit{closure}(\lambda x(e), \rho')v \to \mathit{closure}(\lambda x(e'), \rho')\ v}$$

$$\frac{}{\rho\ \vdash\ \mathit{closure}(\lambda x(v'), \rho')\ v \to v'} \qquad \frac{\rho(x) = v}{\rho\ \vdash\ x \to v}$$

A further imperative extension causes another redefinition of all the previous rules by adding the notion of a store.

$$\begin{array}{l} e ::= \ldots \mid l \texttt{ := } v \\ v ::= \ldots \mid l \mid \mathit{noop} \\ l \in \mathit{Loc},\ \sigma\ \sigma' \in \mathit{Store} = \mathit{Loc} \to \mathit{Val} \end{array}$$

$$\frac{n = n_1 + n_2}{\rho\ \vdash\ \langle n_1 \texttt{ + } n_2, \sigma\rangle \to \langle n, \sigma\rangle}$$

$$\frac{\rho\ \vdash\ \langle e_1, \sigma\rangle \to \langle e_1', \sigma'\rangle}{\rho\ \vdash\ \langle e_1 \texttt{ + } e_2, \sigma\rangle \to \langle e_1' \texttt{ + } e_2, \sigma'\rangle} \qquad \frac{\rho\ \vdash\ \langle e_2, \sigma\rangle \to \langle e_2', \sigma'\rangle}{\rho\ \vdash\ \langle v_1 \texttt{ + } e_2, \sigma\rangle \to \langle v_1 \texttt{ + } e_2', \sigma'\rangle}$$

$$\frac{\rho\ \vdash\ \langle e_1, \sigma\rangle \to \langle e_1', \sigma'\rangle}{\rho\ \vdash\ \langle e_1\ e_2, \sigma\rangle \to \langle e_1'\ e_2, \sigma'\rangle} \qquad \frac{\rho\ \vdash\ \langle e_2, \sigma\rangle \to \langle e_2', \sigma'\rangle}{\rho\ \vdash\ \langle v_1\ e_2, \sigma\rangle \to \langle v_1\ e_2', \sigma'\rangle}$$

$$\frac{}{\rho\ \vdash\ \langle \lambda x(e), \sigma\rangle \to \langle \mathit{closure}(\lambda x(e), \rho), \sigma\rangle}$$

$$\frac{\rho'[x \mapsto v]\ \vdash\ \langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\rho\ \vdash\ \langle \mathit{closure}(\lambda x(e), \rho')v, \sigma\rangle \to \langle \mathit{closure}(\lambda x(e'), \rho')v, \sigma'\rangle}$$

$$\frac{}{\rho\ \vdash\ \langle \mathit{closure}(\lambda x(v'), \rho')v, \sigma\rangle \to \langle v', \sigma\rangle} \qquad \frac{\rho(x) = v}{\rho\ \vdash\ \langle x, \sigma\rangle \to \langle v, \sigma\rangle}$$

$$\rho\ \vdash\ \langle l \texttt{ := } v, \sigma\rangle \to \langle \mathit{noop}, \sigma[l \mapsto v]\rangle \qquad \frac{\sigma(l) = v}{\rho\ \vdash\ \langle l, \sigma\rangle \to \langle v, \sigma\rangle}$$

B The Example Revisited

We give below a modular executable rewriting semantics in Maude for the example language in Section A using the general method of Section 3.1 to exactly mirror the SOS rules. Of course, all the infrastructure of environments and stores has to be made explicit; this is done in the modules `ENV` and `STORE`. In a few places we use some new Maude 2.0 features like: (i) `owise` equations, that can be applied when no other equation with same top function symbol does and abbreviate expanded conditional equations; and (ii) matching equational conditions of the form $t := u$, where t is a constructor term whose variables are instantiated by matching the canonical form of u. Note the choice of `_+'_` for expression addition. Using `_+_` is also possible; in that case subsort overloading would render rule `[1]` unnecessary.

```
mod ADD-EXP is
 extending RCONF .     protecting NAT .
 sorts Val Exp .
 subsorts Nat < Val < Exp < Program .
 op _+'_ : Exp Exp -> Exp .

 vars n1 n2 : Nat .      vars e1 e2 e'1 e'2 : Exp .
 var v1 : Val .          vars PR PR' : PreRecord .

 rl [1] :  {n1 +' n2 , {PR}} => [n1 + n2 , {PR}] .
 crl [2] : {e1 +' e2 , {PR}} => [e'1 +' e2 , {PR'}] if
                                 {e1 , {PR}} => [e'1 , {PR'}] .
 crl [3] : {v1 +' e2 , {PR}} => [v1 +' e'2 , {PR'}] if
                                 {e2 , {PR}} => [e'2 , {PR'}] .
endm

fmod ENV is
 extending RECORD .
 sorts Var Val VarVal EMSet EMap Env .
 subsort VarVal < EMSet .
 subsort EMap < Env < Component .        *** EMap concrete subsort of Env
 op env : -> Index .
 op _[_|->_] : Env Var Val -> Env .      *** abstract interface
 op _(_) : [Env] [Var] -> [Val] .        *** abstract interface
 op [_,_] : Var Val -> VarVal .
 op mt : -> EMSet .
 op _ _ : EMSet EMSet -> EMSet [assoc comm id: mt] .
 op <_> : [EMSet] -> [EMap] .
 op dupl : [EMSet] -> [Bool] .

 var x : Var .  vars rho : Env .  vars v v1 : Val .  var ems : EMSet .

 mb (env : rho) : Field .
 eq dupl([x,v] [x,v1] ems) = true .
```

```
 cmb < ems > : EMap if dupl(ems) =/= true .
 ceq < [x,v] ems > [x |-> v1] =  < [x,v1] ems > if < [x,v] ems > : EMap .
 ceq < ems >[x |-> v1] =  < [x,v1] ems > if dupl(ems) =/= true [owise] .
 ceq < [x,v] ems >(x) = v if < [x,v] ems > : EMap .
endfm

mod FUN-EXP is
 extending ADD-EXP .
 extending ENV .
 sorts Lambda App Closure .
 subsorts Var App Lambda < Exp .
 subsort Closure < Val .
 op __ : Exp Exp -> App .
 op \_(_) : Var Exp -> Lambda .
 op closure : Exp Env -> Val .
 var x : Var .     vars e e' e1 e2 e'1 e'2 : Exp .   vars rho rho' : Env .
 vars v v' v1 : Val .   vars PR PR' : PreRecord .    var ems : EMSet .

 crl [4] : {e1 e2 , {PR}} => [e'1 e2,{PR'}] if {e1 ,{PR}} => [e'1,{PR'}] .
 crl [5] : {v1 e2,{PR}} => [v1 e'2,{PR'}] if {e2,{PR}} => [e'2,{PR'}] .
 rl [6] : {\ x(e),{(env : rho),PR}} => [closure(\ x(e),rho),{(env : rho),PR}] .
 crl [7] : {closure(\ x(e),rho') v,{(env : rho),PR}} =>
           [closure(\ x(e'),rho') v,{(env : rho),PR'}] if
           {e,{(env : rho'[x |-> v]),PR}} => {e',{(env : rho'[x |-> v]),PR'}} .
 rl [8] : {closure(\ x(v'),rho') v,{(env : rho),PR}} => [v',{(env : rho),PR}] .
 crl [9] : {x,{(env : rho), PR}} =>  [v,{(env : rho), PR}] if v := rho(x) .
endm

fmod STORE is
 extending RECORD .
 sorts Loc Val LocVal SMSet SMap Store .
 subsort LocVal < SMSet .
 subsort SMap < Store < Component .         *** SMap concrete subsort of Store
 op _[_|->_] : Store Loc Val -> Store .     *** abstract interface
 op _(_) : [Store] [Loc] -> [Val] .         *** abstract interface
 op st : -> Index .
 op [_,_] : Loc Val -> LocVal .
 op mt : -> SMSet .
 op _ _ : SMSet SMSet -> SMSet [assoc comm id: mt] .
 op <_> : [SMSet] -> [SMap] .
 op dupl : [SMSet] -> [Bool] .

 var l : Loc .   var sigma : Store .   var v v1 : Val . var sms : SMSet .

 mb (st : sigma) : Field .
 eq dupl([l,v] [l,v1] sms) = true .
 cmb < sms > : SMap if dupl(sms) =/= true .
 ceq < [l,v] sms > [l |-> v1] =  < [l,v1] sms > if < [l,v] sms > : SMap .
 ceq < sms >[l |-> v1] =  < [l,v1] sms > if dupl(sms) =/= true [owise] .
 ceq < [l,v] sms >(l) = v if < [l,v] sms > : SMap .
```

```
endfm

mod ASSIGN-EXP is
 extending FUN-EXP .
 extending STORE .
 sorts Assign Noop .
 subsorts Loc Noop < Val .
 subsort Assign < Exp .
 op _:=_ : Loc Val -> Assign .
 op noop : -> Noop .

 var l : Loc .   var sigma : Store .   var v : Val . vars PR : PreRecord .

 rl [10] : {l := v,{(st : sigma), PR}} => [noop,{(st : sigma[l |-> v]), PR}] .
 crl [11] : {l,{(st : sigma),PR}} =>  [v,{(st : sigma),PR}] if v := sigma(l) .
endm
```

C Proof of Theorem 1 (Strong Bisimulation)

We reason by induction on the depth of both the proof tree for an MSOS transition and the proof of a one-step rewrite. In both cases, proofs of purely equational conditions are considered of depth 0. We define the depth of a proof term for a one-step rewrite as follows. Such one-step proof terms are of the form $\mathtt{step}(\theta, \alpha)$, i.e., an application of the `step` rule with θ a substitution and α a proof of a one-step rewrite with a rule r' translating an MSOS rule r. We then define $\mathit{depth}(\mathtt{step}(\theta, \alpha)) = \mathit{depth}(\alpha)$. Likewise, the proof term α is of the form $\alpha = r'(\theta', \beta_1, \ldots, \beta_n, \delta)$, with θ' a substition, the β_i proofs of the rewrites in the condition, and δ a proof of the rule's equational condition cnd'. We then define $\mathit{depth}(r'(\theta', \beta_1, \ldots, \beta_n, \delta)) = 1 + \mathit{max}\{\mathit{depth}(\beta_1), \ldots, \mathit{depth}(\beta_n)\}$. The theorem then follows directly from the following stronger result, which indicates that the strong bisimulation is indeed depth-preserving and therefore involves computations of the same complexity on each side:

For each natural number n and for each pair of states $\langle t, u\rangle$ in $\mathbb{L}_{\mathcal{S}}$ and $\langle t, w\rangle$ in $\mathbb{L}^{\pi}_{\mathbb{T}_{\mathit{Reach}(\tau(\mathcal{S}))}}$ with $w \in \rho^{-1}(u)$ we have:

1. for each transition $\langle t, u\rangle \xrightarrow{v} \langle t', u'\rangle$ in $\mathbb{L}_{\mathcal{S}}$ with a proof of depth n there is a transition $\langle t, w\rangle \xrightarrow{v} \langle t', w'\rangle$ in $\mathbb{L}^{\pi}_{\mathbb{T}_{\mathit{Reach}(\tau(\mathcal{S}))}}$ with a proof of depth n and with $\rho(w') = u'$, and
2. for each transition $\langle t, w\rangle \xrightarrow{v} \langle t', w'\rangle$ in $\mathbb{L}^{\pi}_{\mathbb{T}_{\mathit{Reach}(\tau(\mathcal{S}))}}$ with a proof of depth n there is a transition $\langle t, u\rangle \xrightarrow{v} \langle t', u'\rangle$ in $\mathbb{L}_{\mathcal{S}}$ with a proof of depth n there and with $\rho(w') = u'$.

We prove this stronger result by induction on n. Since there are no proofs of depth 0 in either side, the base case holds trivially. To prove the induction step for (1) and (2) we should reason by cases on the syntactic form of the record expression labeling the conclusion of the rewrite rule r in R. We treat the most complicated case, namely the case in which this expression is of the form $v_0 = \{i_1 : w_1, \ldots, i_n : w_n, PR\}$. To prove (1), suppose the transition $\langle t, u\rangle \xrightarrow{v} \langle t', u'\rangle$ has a proof of depth $n+1$ applying r. This means that there is a ground substitution θ such that $t \xrightarrow{v} t'$ is the substitution instance by θ of the rule's conclusion, say, $t_0 \xrightarrow{v_0} t'_0$, so that we have $\theta(t_0) = t$, and $\theta(t'_0) = t'$. Now notice that for each $w \in \rho^{-1}(u)$ there is a unique w' such that: (i) $v = (w \mapsto w')$ and (ii) $\rho(w') = cod(v) = u'$, namely w' is obtained from w and v by keeping the same read-only fields, taking as read-write fields the primed ones in v, and appending to the value of each write-only field in w the corresponding value in v (appended on the right). Therefore, if there is a transition starting from $\langle t, w\rangle$ and simulating $\langle t, u\rangle \xrightarrow{v} \langle t', u'\rangle$, this transition must be unique. We will be done with the proof of (1) if we show that such transition exists and can be performed with the rule r' that translates rule r. The rule r' is of the form

$$r : \{t_0, v_0^{pre}\} \longrightarrow [t'_0, v_0^{post}] \;\; if \;\; \ldots \wedge \; cnd \; \wedge \; C \sqsubseteq C'$$

where ... abbreviates the rewrites in the conditon, and there are new variables introduced in r' such as: (i) A, B, C in v^{pre}, and A, B', C' in v^{post} in lieu of PR, and a fresh new list variable l_j for each write-only index i_j so that then we have $i_j : l_j$ in v^{pre}, and $i_j : l_j.w_j$ in v^{post}, plus possibly extra new variables introduced for similar reasons in the rule's rewrite conditions. We claim that, given $\langle t, w\rangle = \langle \theta(t_0), w\rangle$, we can obtain a substitution θ' from θ by: (i) assigning to each l_j the value of field i_j in w, (ii) assigning to the variables A, B, B' the fragments of $\theta(PR)$ corresponding to read-only fields, unprimed read-write fields, and primed read-write fields, and (iii) assigning to the variable C the remaining write-only fields in w and to C' the same fields with new values such the condition $\theta'(C) \sqsubseteq \theta'(C')$ is obviously satisfied; similarly, if there were extra variables in the condition of r we can likewise obtain similar assignments in θ' so that the rewrites in the condition of r' when instantiated by θ' are (replacing curly and square brackets by angle brackets throughout) strongly bisimilar to the rewrites in r. By the induction hypothesis we then have rewrite proofs for the θ' instance of r''s rewrite conditions whose maximal depth is n and we get a rewrite proof of depth $n+1$ as desired. Note that we have implictily used the fact that, for the equational condition *cnd* we have $E' \vdash cnd$ iff $E \vdash cnd$. This follows from the the construction of (Σ', E') from (Σ, E) using the assumption that, besides the operations in the extension of `RECORD` (which by the assumption (i) about the MSOS rules R being in normal form cannot apper in *cnd*) no other operations in Σ have kinds in the `RECORD` extension as arguments, except for the `[Component]` kind. The proof of (2) amounts to a detailed description of how we can likewise obtain a substitution θ for r from our original θ' for r' and uses similar arguments.