Pragmatics of Modular SOS

Peter D. Mosses

BRICS Report Series RS-03-52

ISSN 0909-0878 December 2003

Pragmatics of Modular SOS [⋆]

Peter D. Mosses

BRICS[⋆⋆] and Department of Computer Science, University of Aarhus
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark
`pdmosses@brics.dk`, WWW home page: `http://www.brics.dk/~pdm`

Abstract. Modular SOS is a recently-developed variant of Plotkin's Structural Operational Semantics (SOS) framework. It has several pragmatic advantages over the original framework—the most significant being that rules specifying the semantics of individual language constructs can be given *definitively*, once and for all.
Modular SOS is being used for teaching operational semantics at the undergraduate level. For this purpose, the meta-notation for modular SOS rules has been made more user-friendly, and derivation of computations according to the rules is simulated using Prolog.
After giving an overview of the foundations of Modular SOS, this paper gives some illustrative examples of the use of the framework, and discusses various pragmatic aspects.

1 Introduction

Structural Operational Semantics (SOS) [21] is a well-known framework that can be used for specifying the semantics of concurrent systems [1, 9] and programming languages [10]. It has been widely taught, especially at the undergraduate level [7, 20–23]. However, the modularity of SOS is quite poor: when extending a pure functional language with concurrency primitives and/or references, for instance, the SOS rules for all the functional constructs have to be completely reformulated [2].

As the name suggests, Modular SOS (MSOS) [13, 14] is a variant of SOS that ensures a high degree of modularity: the rules specifying the MSOS of individual language constructs can be given *definitively*, once and for all, since their formulation is completely independent of the presence or absence of other constructs in the described language. When extending a pure functional language with concurrency primitives and/or references, the MSOS rules for the functional constructs don't need even the slightest reformulation [17].

In denotational semantics, the problem of obtaining good modularity has received much attention, and has to a large extent been solved by introducing so-called monad transformers [11]. MSOS provides an analogous (but significantly simpler) solution for the structural approach to operational semantics.

[⋆] Invited paper, published in *Proc. AMAST 2002*, LNCS Vol. 2422, pages 21–40. © Springer, 2002.
[⋆⋆] Basic Research in Computer Science (`http://www.brics.dk`), funded by the Danish National Research Foundation

The crucial feature of MSOS is to insist that states are merely abstract syntax and computed values, omitting the usual auxiliary information (such as environments and stores) that they include in SOS. The only place left for auxiliary information is in the labels on transitions. This seemingly minor notational change—coupled with the use of symbolic indices to access the auxiliary information—is surprisingly beneficial. MSOS rules for many language constructs can be specified independently of whatever components labels might have; rules that require particular components can access and set those components without mentioning other components at all. For instance, the MSOS rules for if-expressions do not require labels to have any particular components, and their formulation remains valid regardless of whether expressions are purely functional, have side-effects, raise exceptions, or interact with concurrent processes. Rules specifying the semantics of all individual language constructs can be given definitively in MSOS, once and for all.

MSOS is being used by the author for teaching operational semantics at the undergraduate level (5th semester) [19]. The original, rather abstract notation used for accessing and setting components of labels in MSOS rules [13, 14] has been made more user-friendly, to increase perspicuity. Experimentally, derivation of computations according to MSOS rules is being simulated using Prolog, so that students can trace how the rules shown in examples actually work, and check whether their own rules for further language constructs provide the intended semantics.

Section 2 gives an overview of the foundations of MSOS, recalling the basic notions of transition systems and introducing meta-notation. Section 3 provides illustrative examples of MSOS rules for some constructs taken from Standard ML. Section 4 discusses various pragmatic aspects of MSOS, including tool support and the issue of how to choose between the small-step and big-step ("natural semantics") styles. Section 5 concludes by indicating some topics left for future work.

2 Foundations

This section gives an overview of the foundations of MSOS. It explains the differences between conventional SOS and MSOS, and introduces the meta-notation that will be used for the illustrative examples in Sect. 3. A previous paper on MSOS [13] gives a more formal presentation, focussing on points of theoretical interest.

SOS and MSOS are both based on transition systems. As the name suggests, a transition system has a set of states Q and a set of transitions between states; the existence of a transition from Q_1 to Q_2 is written[1] $Q_1 \longrightarrow Q_2$. A *labelled* transition system has moreover a set of labels X, allowing different transitions

[1] For economy of notation, let us exploit names of sets such as Q also as meta-variables ranging over those sets, distinguishing different meta-variables over the same set by subscripts and/or primes.

$Q_1 \ -X\rightarrow \ Q_2$ (usually written $Q_1 \xrightarrow{X} Q_2$) between Q_1 and Q_2. Labels on transitions are optional in SOS, but obligatory in MSOS.

A *computation* in a transition system is a finite or infinite sequence of composable transitions $Q_i \longrightarrow Q_{i+1}$. With labelled transitions $Q_i \ -X_i\rightarrow \ Q_{i+1}$, the *trace* of the computation is the sequence of the labels X_i. In an *initial* transition system, computations are required to start from initial states I. In a *final* transition system, (finite) computations have to end in final states F, from which there can be no further transitions. Non-final states with no transitions are called *stuck*.

States in SOS and MSOS generally involve both *abstract syntax* (trees) and *computed values*. The abstract syntax of an entire program is used to form the initial state of a computation. Initial states involving parts of programs (declarations, expressions, statements, etc.) are required too, due to the structural nature of SOS and MSOS, whereby the transitions for a compound construct generally depend on transitions for its components. Final states for programs and their parts give computed values: declarations compute binding maps (i.e. environments), expressions compute the expected values, whereas statements compute only a fixed null value.

With the so-called *big-step* (or "natural semantics" [8]) style of SOS and MSOS, computations always go directly from initial states to final states in a single transition. The original *small-step* style favoured by Plotkin[21] requires computations to proceed gradually through intermediate states. Since initial states involve abstract syntax and final states involve computed values, it is unsurprising that intermediate states involve a mixture of these, in the form of abstract syntax trees where some nodes may have been replaced by their computed values. We refer to such mixed trees as *value-added* (abstract) syntax trees; they include pure syntax trees and computed values.

So much for the main features that SOS and MSOS have in common. Let us now consider their differences, which concern restrictions on the structure of states and labels, and whether labels affect the composability of transitions or not:

- In MSOS, states are restricted to value-added abstract syntax trees. Initial states are therefore pure abstract syntax, and final states are simply computed values. SOS, in contrast, allows states to include auxiliary components such as stores and environments.
- MSOS requires labels to be records (i.e. total maps on finite sets of indices, also known as indexed products). The components of these records are unrestricted, and usually include the auxiliary entities that would occur as components of states in SOS. Labels are often omitted altogether in SOS descriptions of sequential languages.
- In MSOS, transitions are composable only when their labels are composable, as described below. In SOS, composability of transitions is independent of their labels, and depends only on the states involved.

Each component of labels in MSOS has to be classified as inherited, variable, or observable. The classification affects composability of labels, as follows:

– When a component is classified as *inherited*, two labels are composable only when the value of this component is the same in both labels.
– A component classified as *variable* has both an initial and a final value in each label, and two labels X_1, X_2 are composable only when the final value of the component in X_1 is the same as its initial value in X_2. When the index for the initial value of the component is i, the index for the final value is written i'.
– The possible values of a component classified as *observable* form a monoid (e.g. lists under concatenation). Its actual values do not affect composability.

When two labels X_1, X_2 are composable, their composition is determined in the obvious way, and written $X_1;X_2$.

A further significant difference between SOS and MSOS is that MSOS provides a built-in notion of *unobservable transition*, characterized simply by the components of the label: each variable component must remain constant, and each observable component must be the unit of the corresponding monoid (e.g. the empty list). The distinction between arbitrary labels X and labels U for unobservable transitions is crucial when formulating specifications in MSOS; it may also be used to define so-called observational equivalence.

Readers who are familiar with Category Theory may like to know that labels in MSOS are indeed the morphisms of a category, with the unobservable labels as identity morphisms (corresponding to the objects of the category, which are not referred to directly). Simple functors called basic label transformers can be used for adding new components to labels [13, 14]. Surprisingly, it appears that this straightforward use of categories of labels had not previously been exploited in work on transition systems.

One of the most distinctive features of SOS is the way that axioms and inference rules (together referred to simply as *rules*) are used to specify transition systems that represent the semantics of programming languages. The premises and conclusions of the rules are assertions of transitions $t_1 \ {-t}{\rightarrow}\ t_2$ or $t_1 \longrightarrow t_2$, where t, t_1, t_2 are terms that may involve constructor operations, auxiliary operations, and meta-variables. Rules may also have side-conditions, which are often equations or applications of auxiliary predicates. It is common practice to write the side-conditions together with the premises of rules, and one may also use rules to defined auxiliary operations and predicates, so the general form of a rule is:

$$\frac{c_1, \ldots, c_n}{c} \tag{1}$$

where each of $c, c_1, \ldots, c_n (n \geq 0)$ may be a transition $t_1 \ {-t}{\rightarrow}\ t_2$ or $t_1 \longrightarrow t_2$, an equation $t_1 = t_2$, or an application $p(t_1, \ldots, t_k)$ of a predicate p. Provided that the rules do not involve negations of assertions in the premises, they can be interpreted straightforwardly as inductive definitions of relations, so that assertions are satisfied in models if and only if they follow from the rules by ordinary logical inference. (The situation when negative premises are allowed is quite complicated [1], and not considered further here.)

Rules in MSOS have the same general form as in SOS. The rules specifying transitions for a programming construct are typically *structural*, having conclusions of the form $c(m_1, \ldots, m_k) \ -m\rightarrow \ t$ where c is a constructor and $m, m_1, \ldots, m_k$ are simply meta-variables, and with premises involving one or more assertions of the form $m_i \ -m'\rightarrow \ t'$; but note that being structural is not essential for a set of rules to specify a well-defined transition system.

MSOS provides some operations for records, finite maps, and lists; the notation corresponds closely to that used in SML and (for lists) in Prolog:

- A record with components $t_1, \ldots, t_n (n \geq 0)$ indexed respectively by $i_1, \ldots, i_n$ is written $\{i_1{=}t_1, \ldots, i_n{=}t_n\}$; the value is independent of the order in which the components are written, and the indices must be distinct. When t' evaluates to a record not having the index i, the term $\{i{=}t|t'\}$ evaluates to that record extended with the value of t indexed by i (this notation is by analogy with Prolog notation for lists, and should not be confused with set comprehension). When i is a particular index, and t and t' are simply meta-variables, the equation $t'' = \{i{=}t|t'\}$ can be used to extract the value of the component indexed i from the record given by t'', with t' being the rest of the record—without mentioning what other indices might be used in the record. A special dummy meta-variable written '...' may be used in place of t' when the rest of the rule does not refer to t'; an alternative is to use $t''.i$ to select the component indexed i from t''.
- The above notation for records may also be used for constructing finite (partial) maps between sets, and for selecting values and the remaining maps. Two further operations are provided: t/t' is the map with t overriding t'; and $dom(t)$ gives the domain of the map t (i.e. a set).
- A list with components $t_1, \ldots, t_n (n \geq 0)$ is written $[t_1, \ldots, t_n]$.

One final point concerning rules: they can only be instantiated when all terms occurring in them have defined values. (Some of the operations used in terms may be partial, giving rise to the possibility of their applications having undefined values.) Note that applications of operations are never defined, and applications of predicates never hold, when any argument term has an undefined value.

3 Illustrative Examples

The combination of the rules given in this section provides an MSOS for a simple sub-language of Standard ML (SML). The concrete syntax of the described language is indicated by the grammar in Table 1 for declarations D, expressions E, operators O, and identifiers I. The example constructs have been chosen so as to illustrate various features of MSOS. The MSOS rules are formulated in terms of abstract syntax constructor operations whose correspondence to the concrete constructs is rather obvious.

A set of rules specifying the MSOS of a particular construct generally makes requirements on the components of labels and on various sets of values. For instance, it might be required for some construct that labels have a component

Table 1. Concrete syntax for the examples

$$
\begin{array}{ll}
D ::= & \texttt{val}\ I\ \texttt{=}\ E \mid \texttt{rec}\ D \\
E ::= & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \mid \texttt{(}E\texttt{,}E\texttt{)} \mid \\
& E\ O\ E \mid \texttt{true} \mid \texttt{false} \mid [\texttt{0}-\texttt{9}]^{+} \mid \\
& I \mid \texttt{let}\ D\ \texttt{in}\ E\ \texttt{end} \mid \texttt{fn}\ I\ \texttt{=>}\ E \mid E\ E \mid \\
& \texttt{ref}\ E \mid E\ \texttt{:=}\ E \mid \texttt{!}\ E \mid E\texttt{;}E \mid \texttt{while}\ E\ \texttt{do}\ E \mid \\
& \texttt{raise}\ E \mid E\ \texttt{handle}\ \texttt{x=>x} \\
O ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{=} \mid \texttt{>} \\
I\ \ ::= & [\texttt{a}-\texttt{z}]^{+}
\end{array}
$$

referred to as *bindings*, giving a map B from identifiers to values V, and that the set of values includes abstractions (representing functions). Such requirements are indicated informally below, in the interests of focussing attention on the rules themselves; the collected requirements imposed by all the constructs are listed at the end of the section.

All the rules given here are in the so-called *small-step* style. In fact MSOS does support the use of the big-step "natural semantics" style, as well as mixtures of the two styles. The general use of the small-step style will be motivated in Sect. 4.

3.1 Simple Expressions

The rules for the operational semantics of if-expressions in Table 2 require that expression values V include the truth-values *true* and $false$. The use of the

Table 2. If-expressions: `if` E `then` E `else` E

$$E ::= \mathit{if}(E, E, E)$$

$$\frac{E_1 \ -X\!\!\rightarrow E_1'}{\mathit{if}(E_1, E_2, E_3) \ -X\!\!\rightarrow \mathit{if}(E_1', E_2, E_3)} \tag{2}$$

$$\mathit{if}(\mathit{true}, E_2, E_3) \ -U\!\!\rightarrow E_2 \tag{3}$$

$$\mathit{if}(\mathit{false}, E_2, E_3) \ -U\!\!\rightarrow E_3 \tag{4}$$

variable X ranging over arbitrary labels in rule (2) reflects the fact that whatever information is processed by any step of E_1, it is exactly the same as that processed by the corresponding step of $\mathit{if}(E_1, E_2, E_3)$. In contrast, the variable U in rule (3) and rule (4) ranges only over labels on unobservable steps, which are merely internal changes to configurations without any accompanying information processing. We henceforth suppress such single occurrences of U in rules, writing $\ldots \longrightarrow \ldots$ instead of $\ldots -U\!\!\rightarrow \ldots$.

Clearly, rule (2) allows the computation of $\mathit{if}(E_1, E_2, E_3)$ to start with a computation of E_1, and a computation of E_2 or E_3 can start only when that of E_1 terminates with computed value *true*, respectively *false*.

The rules in Table 3 require that the expressible values V are closed under formation of pairs (V_1, V_2). Rule (6) allows the computation of E_2 to start only

Table 3. Pair-expressions: `(`E`,`E`)`

$$E ::= \mathit{pair}(E, E)$$

$$\frac{E_1 \; {-X\!\!\rightarrow} \; E_1'}{\mathit{pair}(E_1, E_2) \; {-X\!\!\rightarrow} \; \mathit{pair}(E_1', E_2)} \tag{5}$$

$$\frac{E_2 \; {-X\!\!\rightarrow} \; E_2'}{\mathit{pair}(V_1, E_2) \; {-X\!\!\rightarrow} \; \mathit{pair}(V_1, E_2')} \tag{6}$$

$$\mathit{pair}(V_1, V_2) \longrightarrow (V_1, V_2) \tag{7}$$

after E_1 has computed a value V_1, thus reflecting sequential evaluation of the pair of expressions (following SML); replacing V_1 in that rule by E_1 would allow arbitrary interleaving of the steps of the computations of E_1 and E_2.

The rules for binary operations shown in Table 4 make use of *operate*, which is defined in Table 5. For the latter, it is required that the expressible values V include both (integer) numbers N and the truth-values.

Table 4. Binary operation expressions: $E \; O \; E$

$$E ::= \mathit{binary}(O, E, E)$$

$$\frac{E_1 \; {-X\!\!\rightarrow} \; E_1'}{\mathit{binary}(O, E_1, E_2) \; {-X\!\!\rightarrow} \; \mathit{binary}(O, E_1', E_2)} \tag{8}$$

$$\frac{E_2 \; {-X\!\!\rightarrow} \; E_2'}{\mathit{binary}(O, V_1, E_2) \; {-X\!\!\rightarrow} \; \mathit{binary}(O, V_1, E_2')} \tag{9}$$

$$\frac{\mathit{operate}(O, N_1, N_2) = V}{\mathit{binary}(O, N_1, N_2) \longrightarrow V} \tag{10}$$

Table 6 is concerned with the evaluation of literal truth-values and numbers. Since the details of how to evaluate a sequence of (decimal) digits to the corresponding natural number are of no interest, let us assume that this is done already during the passage from concrete to abstract syntax, so that N is actually an integer.

Table 5. Binary operations: `+` | `-` | `*` | `=` | `>`

$$O ::= \mathit{plus} \mid \mathit{minus} \mid \mathit{times} \mid \mathit{equals} \mid \mathit{greater}$$

$$\mathit{operate}(\mathit{plus}, N_1, N_2) = N_1 + N_2 \tag{11}$$

$$\mathit{operate}(\mathit{minus}, N_1, N_2) = N_1 - N_2 \tag{12}$$

$$\mathit{operate}(\mathit{times}, N_1, N_2) = N_1 * N_2 \tag{13}$$

$$\mathit{operate}(\mathit{equals}, N_1, N_2) = \mathit{true}\ \mathit{when}\ N_1 = N_2\ \mathit{else}\ \mathit{false} \tag{14}$$

$$\mathit{operate}(\mathit{greater}, N_1, N_2) = \mathit{true}\ \mathit{when}\ N_1 > N_2\ \mathit{else}\ \mathit{false} \tag{15}$$

Table 6. Literal truth-values and numbers: `true`|`false`|$[0-9]^+$

$$E ::= \mathit{lit}(\mathit{true}) | \mathit{lit}(\mathit{false}) | \mathit{num}(N)$$

$$\mathit{lit}(\mathit{true}) \longrightarrow \mathit{true} \tag{16}$$

$$\mathit{lit}(\mathit{false}) \longrightarrow \mathit{false} \tag{17}$$

$$\mathit{num}(N) \longrightarrow N \tag{18}$$

3.2 Bindings

So far, none of the rules have referred to any particular components of labels: the set of labels has been left completely open, and could even be just a singleton. Rule (19) in Table 7, however, requires that the labels have at least a component referred to as *bindings*. Let this component be "inherited", and let its values be maps B from identifiers I to values V. (We do not bother here to make the usual notational distinction between the set of bindable—also known as denotable—values and the set V of expressible values, since for our illustrative constructs they happen to coincide.)

Table 7. Value-identifiers: I

$$E ::= \mathit{ide}(I)$$

$$\frac{B = U.\mathit{bindings},\ V = B.I}{\mathit{ide}(I) \ {-U\!\!\rightarrow}\ V} \tag{19}$$

Rule (19) specifies that an identifier I evaluates to the value V to which it is bound by the bindings map B provided by the label U. Notice that if I is not in $\mathit{dom}(B)$, the rule simply cannot be applied, since one of its conditions isn't satisfied. (Equations such as $B = U.\mathit{bindings}$ should be regarded formally as side-conditions, but it is notationally convenient to list them together with whatever premises the rule might have.)

This next illustration is particularly important. The rules in Table 8 show how block structure and nested scopes of declarations are specified in MSOS. Just as expressions E compute values V, a declaration D computes a binding map B, mapping the identifiers declared by D to values V.

Table 8. Let-expressions: `let` D `in` E `end`

$$E ::= \mathit{let}(D, E)$$

$$\frac{D \ {-X\!\!\rightarrow}\ D'}{\mathit{let}(D, E)\ {-X\!\!\rightarrow}\ \mathit{let}(D', E)} \tag{20}$$

$$\frac{\begin{array}{c}X = \{\mathit{bindings}{=}B_1|X'\},\ B_2 = B/B_1,\\ X'' = \{\mathit{bindings}{=}B_2|X'\},\ E\ {-X''\!\!\rightarrow}\ E'\end{array}}{\mathit{let}(B, E)\ {-X\!\!\rightarrow}\ \mathit{let}(B, E')} \tag{21}$$

$$\mathit{let}(B, V) \longrightarrow V \tag{22}$$

The computation of $\mathit{let}(B, E)$ continues with that of E. The equations given as conditions in rule (21) ensure that the label X on a step for $\mathit{let}(B, E)$ has exactly the same components as the label X'' on a step for E, except of course for the *bindings* component, which is B_1 in X but B/B_1 (B overriding B_1) in X''. Recall that an equation such as $X = \{\mathit{bindings}{=}B_1|X'\}$ both identifies the *bindings* component as B_1 and the record consisting of all the other fields of X as X'.

If the computation of E finally computes a value V, that is also the value computed by the let-expression, as specified by rule (22).

Table 9. Value-declarations: `val` I `=` E

$$D ::= \mathit{value}(I, E)$$

$$\frac{E\ {-X\!\!\rightarrow}\ E'}{\mathit{value}(I, E)\ {-X\!\!\rightarrow}\ \mathit{value}(I, E')} \tag{23}$$

$$\mathit{value}(I, V) \longrightarrow \{I{=}V\} \tag{24}$$

The rules for (non-recursive) value declarations are given in Table 9, and are completely straightforward. Closely related to value declarations are value abstractions and (call by value) applications, which are specified in Tables 10 and 11.

The value computed by an abstraction is a so-called closure, which is conveniently represented by using a let-expression to attach the current bindings B to the abstraction itself. If we didn't already have the constructor for let-expressions

Table 10. Value-abstractions: `fn` I `=>` E

$$E ::= \mathit{abstraction}(I, E)$$

$$\frac{U = \{\mathit{bindings}{=}B|\ldots\}}{\mathit{abstraction}(I, E) \ {-U\rightarrow}\ \mathit{abs}(\mathit{let}(B, \mathit{abstraction}(I, E)))} \tag{25}$$

available, we would have to define it as auxiliary notation (or provide some other operation allowing bindings to be attached to abstractions). The auxiliary operation $\mathit{abs}(E)$ constructs a value from an arbitrary expression E—we cannot use the expression E itself as a value, since values should always be final states for computations.

Table 11. Abstraction-applications: $E\ E$

$$E ::= \mathit{application}(E, E)$$

$$\frac{E_1 \ {-X\rightarrow}\ E_1'}{\mathit{application}(E_1, E_2) \ {-X\rightarrow}\ \mathit{application}(E_1', E_2)} \tag{26}$$

$$\frac{E_2 \ {-X\rightarrow}\ E_2'}{\mathit{application}(V_1, E_2) \ {-X\rightarrow}\ \mathit{application}(V_1, E_2')} \tag{27}$$

$$\frac{V_1 = \mathit{abs}(\mathit{let}(B, \mathit{abstraction}(I, E)))}{\mathit{application}(V_1, V_2) \longrightarrow \mathit{let}(B, \mathit{let}(\{I{=}V_2\}, E))} \tag{28}$$

Once E_1 has been evaluated to an abstraction value V_1, and E_2 to an argument value V_2, rule (28) replaces $\mathit{application}(V_1, V_2)$ by the appropriate expression, and the computation may continue. Assuming that all the identifiers free in E are bound by B, the application-time bindings provided by U are not used (directly) in that computation: the rules have provided static scopes for bindings.

The constructs that we have specified so far allow the declaration of (call by value) functions, using a combination of value-declarations and value-abstractions, but they do not (directly) support recursive function declarations. The construct $\mathit{recursive}(D)$ specified in Table 12 has the effect of making ordinary function declarations D recursive (it could easily be extended to mutually-recursive declarations).

The auxiliary operation $\mathit{unfold}(D, B)$ is defined by the (equational) rule (31). It returns a binding map where D has been inserted at the appropriate place in the closure, using a further level of let-expression. (It isn't intended for use on bindings of non-function values such as numbers and pairs, but for completeness, rule (32) specifies that it would have no effect on them.) Rule (33) specifies the result of an application when the abstraction incorporates a recursive declaration

Table 12. Recursive declarations: `rec` D

$$D ::= \mathit{recursive}(D)$$

$$\frac{D \;{-X\rightarrow}\; D'}{\mathit{recursive}(D) \;{-X\rightarrow}\; \mathit{recursive}(D')} \tag{29}$$

$$\frac{B' = \mathit{unfold}(\mathit{recursive}(B), B)}{\mathit{recursive}(B) \longrightarrow B'} \tag{30}$$

$$\frac{V = \mathit{abs}(\mathit{let}(B, E)),\; V' = \mathit{abs}(\mathit{let}(B, \mathit{let}(\mathit{recursive}(B_1), E)))}{\mathit{unfold}(\mathit{recursive}(B_1), \{I{=}V\}) = \{I{=}V'\}} \tag{31}$$

$$\frac{V \neq \mathit{abs}(\ldots)}{\mathit{unfold}(\mathit{recursive}(B_1), \{I{=}V\}) = \{I{=}V\}} \tag{32}$$

$$\frac{V_1 = \mathit{abs}(\mathit{let}(B, \mathit{let}(\mathit{recursive}(B_1), \mathit{abstraction}(I, E))))}{\mathit{application}(V_1, V_2) \longrightarrow \mathit{let}(B, \mathit{let}(\mathit{recursive}(B_1), \mathit{let}(\{I{=}V_2\}, E)))} \tag{33}$$

$\mathit{recursive}(B_1)$; the actual unfolding of the recursive declaration is left to the (first step of) the subsequent computation.

Rules for declarations analogous to those illustrated here could be given in conventional SOS, using transitions of the form $B \vdash E \longrightarrow E'$ (after defining a suitable notion of composition for such transitions). The main pragmatic advantages of our MSOS rules will become apparent only in the next sections, where we shall allow expressions to have side-effects and to raise exceptions, without any reformulation of the rules given so far.

3.3 Stores

The imperative constructs described in this section are taken almost unchanged from SML, and involve so-called references to values. References correspond to simple variables, and can be created, updated, and dereferenced. The set of created references, together with the values to which they refer, is represented by a store S mapping locations L to values V. A location is itself a value, and can be stored, as well as bound to identifiers.

Some of the rules given here require that labels X have not only a *store* component, giving the store at the beginning of a transition labelled X, but also a store' component, giving the (possibly different) store at the end of the transition. When a transition labelled X_1 is followed immediately by a transition labelled X_2 in a computation, the store' component of X_1 has to be the same as the *store* component of X_2. Moreover, the *store* and store' components of an unobservable transition labelled U have to be the same as well.

Table 13 gives the rules for reference creation, which is combined with initialization. L in rule (35) can be any location that is not already in use in S. Notice the use of the variable U, which ensures that the extension of S by the association of L with V is the only observable effect of the transition labelled X.

Table 13. Reference-expressions: `ref` E

$$E ::= \mathit{reference}(E)$$

$$\frac{E \ {-X\rightarrow}\ E'}{\mathit{reference}(E) \ {-X\rightarrow}\ \mathit{reference}(E')} \tag{34}$$

$$\frac{X = \{\mathit{store}{=}S, \mathit{store}'{=}S'|U\},\ L \notin \mathit{dom}(S),\ S' = \{L{=}V|S\}}{\mathit{reference}(V) \ {-X\rightarrow}\ L} \tag{35}$$

Table 14. Assignment-expressions: E `:=` E

$$E ::= \mathit{assignment}(E, E)$$

$$\frac{\mathit{assignment}(E_1, E_2) \ {-X\rightarrow}\ \mathit{assignment}(E_1', E_2)}{E_1 \ {-X\rightarrow}\ E_1'} \tag{36}$$

$$\frac{\mathit{assignment}(L_1, E_2) \ {-X\rightarrow}\ \mathit{assignment}(L_1, E_2')}{E_2 \ {-X\rightarrow}\ E_2'} \tag{37}$$

$$\frac{X = \{\mathit{store}{=}S, \mathit{store}'{=}S'|U\},\ L_1 \in \mathit{dom}(S),\ S' = \{L_1{=}V_2|S\}}{\mathit{assignment}(L_1, V_2) \ {-X\rightarrow}\ ()} \tag{38}$$

The rules for assignment given in Table 14 correspond closely to those in Table 13, except that the assignment computes the null value (). Table 15 gives the rules for (explicit) dereferencing; and Table 16 describes expression sequencing, where the value of the first expression is simply discarded. The rule for while-expressions in Table 17 is standard in (small-step) SOS, involving both an if-expression and a sequence-expression (a direct description not involving other constructs seems to be elusive).

Table 15. Dereferencing-expressions: `!` E

$$E ::= \mathit{dereference}(E)$$

$$\frac{E \ {-X\rightarrow}\ E'}{\mathit{dereference}(E) \ {-X\rightarrow}\ \mathit{dereference}(E')} \tag{39}$$

$$\frac{U = \{\mathit{store}{=}S|\ldots\},\ S = \{L{=}V|\ldots\}}{\mathit{dereference}(L) \ {-U\rightarrow}\ V} \tag{40}$$

Despite the fact that the illustrated imperative programming constructs allow expressions to have (side-)effects, no reformulation at all is required for the rules given in the preceding section: they remain well-formed and continue to describe the intended semantics. This is in marked contrast with conventional

Table 16. Sequence-expressions: $E\,;E$

$$E ::= \mathit{sequence}(E, E)$$

$$\frac{E_1 \;-X\!\rightarrow\; E_1'}{\mathit{sequence}(E_1, E_2) \;-X\!\rightarrow\; \mathit{sequence}(E_1', E_2)} \tag{41}$$

$$\mathit{sequence}(V_1, E_2) \longrightarrow E_2 \tag{42}$$

Table 17. While-expressions: `while` E `do` E

$$E ::= \mathit{while}(E_1, E_2)$$

$$\begin{gathered}\mathit{while}(E_1, E_2) \longrightarrow \\ \mathit{if}(E_1, \mathit{sequence}(E_2, \mathit{while}(E_1, E_2)), ())\end{gathered} \tag{43}$$

SOS, where stores would be added explicitly to configurations, requiring all transitions $B \vdash E \longrightarrow E'$ to be reformulated as something like $B \vdash (E, S) \longrightarrow (E', S')$. Actually, the "store convention" adopted in the Definition of SML [10] would avoid the need for such a reformulation, but it has a rather informal character; the incorporation of stores as components of labels in MSOS achieves a similar result, completely formally.

3.4 Exceptions

The description of exception raising and handling in conventional SOS usually involves giving further rules for all the *other* constructs in the language being described, allowing exceptions raised in any component to be "propagated" upwards until an exception handler is reached. Such rules are quite tedious, and undermine modularity. The "exception convention" adopted in the Definition of SML [10] allows the exception propagation rules to be left implicit, but has the disadvantage that the set of presented rules has to be expanded considerably to make the propagation rules explicit before they can be used for deriving computations, etc.

Fortunately, a technique recently proposed by Klin (a PhD student at BRICS in Aarhus) avoids the need for exception propagation rules altogether, and gives excellent modularity. The key idea is illustrated in Table 18 and Table 19: labels on transitions are required to have a further component that simply indicates whether an exception is being raised by the transition or not, and if so, gives the value associated with the raised exception. An exception handler monitors the label of every step of the computation of its body, checking the extra component.

To focus attention on the main features of the new technique, and for brevity, only a simplified indiscriminate exception handler is described here; it is however straightforward to describe SML's exception handlers, which involve pattern-matching.

Table 18. Exception-raising: `raise` E

$$E ::= \mathit{raise}(E)$$

$$\frac{E \;{-X\rightarrow}\; E'}{\mathit{raise}(E) \;{-X\rightarrow}\; \mathit{raise}(E')} \tag{44}$$

$$\frac{X = \{\mathit{raising}{=}[V] | U\}}{\mathit{raise}(V) \;{-X\rightarrow}\; ()} \tag{45}$$

Table 19. Exception-handling: E `handle x=>x`

$$E ::= \mathit{handle}(E)$$

$$\frac{E \;{-X_1\rightarrow}\; E',\; X_1 = \{\mathit{raising}{=}[V] | X'\},\; X = \{\mathit{raising}{=}[\,] | X'\}}{\mathit{handle}(E) \;{-X\rightarrow}\; V} \tag{46}$$

$$\frac{E \;{-X\rightarrow}\; E',\; X = \{\mathit{raising}{=}[\,] | \ldots\}}{\mathit{handle}(E) \;{-X\rightarrow}\; \mathit{handle}(E')} \tag{47}$$

$$\mathit{handle}(V) \longrightarrow V \tag{48}$$

Let us use a list $[V]$ with the single component V to indicate the raising of an exception with value V, and the empty list $[\,]$ to indicate the absence of an exception. Transitions that raise exceptions are always observable.

In Table 18, rule (45) merely sets the *raising* component of X to the list $[V]$. When the raised exception occurs during the computation of $\mathit{handle}(E)$, rule (46) detects the exception and abandons the computation of E, giving V as the computed value. The *raising* component of the label X is set to the empty list, to reflect that the exception has now been handled at this level. The complementary rule (47) deals with a normal step of the computation of E where no exception is raised.

Suppose however that an exception is raised with no enclosing handler. Then the step indicated in rule (45) is completed, and the computation continues normally, as if the raise-expression had merely computed the null value (), although the raising of the exception is observable from the label of that step. To get the intended effect that an unhandled exception should stop the entire program, the entire program should always be enclosed in an extra *handle*.

The technique illustrated here is not restricted to MSOS: it could be used in conventional (small-step) SOS as well. However, in a conventional SOS of a programming language, transitions are usually unlabelled, and the amount of reformulation that would be required to add labels may be a considerable disincentive to exploiting the new technique. In MSOS, transitions are always labelled, and adding a new component to labels doesn't require any reformulation of rules.

3.5 Interaction

Lack of space precludes illustration here of the MSOS of constructs for interactive input and output, process spawning, message passing, and synchronization. An MSOS for the core of Concurrent ML has been given (using a more abstract notation) in a previous paper [17]: the rules are quite similar in style to the conventional SOS rules for concurrent processes, where configurations are generally syntactic, and transitions are labelled, just as in MSOS. What is remarkable with the MSOS rules is that allowing expressions to spawn processes and communicate with other processes requires no reformulation at all of rules given for purely functional expressions.

3.6 Summary of Requirements

The requirements made by the rules given in this section are summarized in Table 20.

Table 20. Summary of Requirements

Values:

$$V ::= \mathit{true} \mid \mathit{false} \mid N \mid (V, V) \mid \mathit{abs}(E) \mid L \mid ()$$
$$B ::= \mathit{Map}(I, V)$$
$$S ::= \mathit{Map}(L, V)$$

Labels:

$$X ::= \{\ \textbf{inherited}\ \ \mathit{bindings} : B;\ \ \textbf{variable}\ \ \mathit{store}, \mathit{store}' : S;\ \ \textbf{observable}\ \mathit{raising} : \mathit{List}(V)\}$$

4 Pragmatic Aspects

The illustrative examples given in the preceding section should have given a reasonable impression of how MSOS works. Let us now consider some important pragmatic aspects of MSOS, concerning modularity, tool support, and the choice between big-step and small-step rules.

4.1 Modularity

The main pragmatic advantage of MSOS over conventional SOS is that the rules describing the intended semantics for each programming construct can be given *definitively*, once and for all. When gradually building up a language, it is never necessary to go back and reformulate previously-given rules to accommodate the addition of new constructs. Moreover, when different languages include the same constructs (e.g. SML and Java both have if-expressions), the MSOS rules describing the semantics of the common constructs may be reused verbatim.

Such features justify the use of the word "modular" in connection with the MSOS framework. The benefits of this kind of modularity are especially apparent when using MSOS to teach operational semantics, and one may hope that they might even encourage language designers to use formal semantics to record their decisions during a design process.

4.2 Tool Support

Another pragmatic aspect that is particularly important for applications of formal semantics in teaching and language design is tool support [6]. This is needed both for developing and browsing (large) semantic descriptions, as well as for validating descriptions by running programs according to their specified semantics. Some tool support for MSOS has already been developed by Braga et al. [3, 4] using the Maude system for Rewriting Logic, but much remains to be done.

In fact it is quite easy to write MSOS rules directly in Prolog. Some examples are given in Table 21, and a Prolog program implementing all the MSOS rules given in this paper is available at `http://www.brics.dk/~pdm/AMAST-02/`.[2] Both records and finite maps are conveniently represented in Prolog as lists of equations, and the Prolog predicates `member(M,L)` and `select(M,L1,L2)` are used to implement conditions concerning labels, bindings, stores, etc. Unary predicates are used to check that variables range only over the intended sets, e.g., `unobs(U)` ensures that `U` is indeed an unobservable label, and `val(V)` checks that `V` is a computed values.

A definite clause grammar for the described language allows programs to be written in concrete syntax, and mapped directly to the abstract syntax constructors used in the rules. The user may determine some of the components of the label on the first transition, and select which components are to be shown at the end of the computation. When the rules are non-deterministic (e.g. describing interleaving or concurrency) the user may investigate successive solutions. The efficiency of modern Prolog implementations (the author uses SWI-Prolog, which is freely available from `http://www.swi-prolog.org/` for most popular platforms) is such that running small test programs using the clauses corresponding to their MSOS rules takes only a few seconds on a typical lap-top.

The Prolog clauses have just as good modularity as the MSOS rules. Tracers and debuggers (SWI-Prolog provides one with an efficient graphical user interface) allow the user to follow changes to configurations at the top level, as well as in-depth derivations of individual steps.

At the time of writing this paper, it remains to be seen whether students will find the Prolog implementation useful for getting a good operational understanding of the MSOS rules, and whether they will find it easy to extend it with implementations of their own rules.

[2] The author has had little experience of programming in Prolog, and would appreciates suggestions for improvements to the code.

Table 21. Examples of MSOS rules in Prolog

```
pair(E1,E2) ---X---> pair(E1_,E2)  :-
      E1 ---X---> E1_.
pair(EV1,E2) ---X---> pair(EV1,E2_) :-
      val(EV1),
      E2 ---X---> E2_.
pair(EV1,EV2) ---U---> (EV1,EV2) :-
      val(EV1), val(EV2), unobs(U).

assignment(E1,E2) ---X---> assignment(E1_,E2) :-
      E1 ---X---> E1_.
assignment(L1,E2) ---X---> assignment(L1,E2_) :-
      val(L1), E2 ---X---> E2_.
assignment(L1,V2) ---X---> null :-
      val(L1), val(V2),
      select(store=S,X,X_), select(L1=_,S,S_),
      select(store_=[L1=V2|S_],X_,U), unobs(U).
```

4.3 Why Not Big Steps?

All the MSOS rules illustrated in Sect. 3 are small-step rules: each step that a construct can take depends on at most one of its components taking a corresponding step. However, MSOS does allow big-step rules as well, where a step represents an entire sub-computation, going straight to a final state—often depending on all its components taking corresponding steps.

For example, consider the big-step rule for pair-expressions given in Table 22. Label composition, written X_1;X_2, has to be used explicitly in big-step rules (recall that in small-step MSOS, computations require adjacent labels to be composable, but composition is not needed in the rules themselves). The specified order of composition of labels indicates that the corresponding computation steps may be taken in the same order.

Table 22. Big-step pair-expressions: `(`E`,`E`)`

$$E ::= \mathit{pair}(E, E)$$

$$\frac{E_1 \ {-X_1\rightarrow}\ V_1,\ E_2 \ {-X_2\rightarrow}\ V_2}{\mathit{pair}(E_1, E_2) \ {-X_1;X_2\rightarrow}\ (V_1, V_2)} \tag{49}$$

Given that MSOS supports both small-step and big-step rules, which should one prefer? Changing from the one style of rule to the other requires major reformulation, so it is important to make a good choice and stick to it.

In his Aarhus lecture notes on SOS [21], Plotkin generally used small-step rules—although by employing the transitive closure of the transition relation in the conditions of some rules, he obtained the effect of mixing the two styles of rules (e.g. a single small step for a statement depended on complete sequences of small steps for its component expressions). Moreover, much of the work on semantics of concurrent processes (CCS, CSP, etc.) is based on small-step rules.

On the other hand, Kahn [8] has advocated exclusive use of big-step rules, and this style was adopted by Milner et al. for the Definition of SML [10]. The pure big-step style has also been widely used in specifying type systems and static semantics.

This author's view is that big-step rules should be used *only* for constructs whose computations always terminate normally: no divergence, no raising of exceptions. This includes literal constants and types, but excludes almost all other programming constructs. (As usual, we are focussing here on dynamic semantics; for static semantics, the recommendation would be to use big-step rules for all constructs.) The reason for the restriction to terminating computations is that big-step semantics is usually based on finite derivations, and non-terminating computations get completely ignored. Regarding exceptions, big-step rules appear to be inherently non-modular: adding exceptions to the described language requires adding rules specifying propagation of exceptions. (The novel technique illustrated in Sect. 3.4 works only for small-step rules.)

It seems to be a widely-held belief that the description of interleaving constructs actually requires small-step rules. Suppose however that one makes a slight generalization to MSOS, allowing not only single records X but also arbitrary (finite) sequences of records $X_1 \ldots X_n$ as labels on transitions. It turns out that big-step rules for interleaving can then be given quite straightforwardly, by allowing the sequences in the labels to be arbitrarily interleaved (i.e., "shuffled"), and restricting the labels on transitions for the entire program to be composable sequences. This technique appears to depend entirely on the use of MSOS: in SOS, the inclusion of auxiliary entities in states makes it awkward (if not impossible) to give an analogous treatment of interleaving. In view of the arguments given above in favour of small-step rules, however, the development of techniques for use in big-step rules is not so interesting from a pragmatical point of view.

5 Conclusion

This paper has given an overview of the foundations of MSOS, and has illustrated the use of MSOS to describe some simple functional and imperative programming constructs. It has also presented a novel technique for the modular description of exception-handling. Finally, it has discussed particular pragmatic aspects such as modularity, tool support, and the choice between small- and big-step rules.

The development of MSOS was originally stimulated by the author's dissatisfaction with the lack of modularity in his own SOS for Action Notation, the semantic notation used in Action Semantics [12]. An MSOS for Action Notation has been given [15] using CASL, the Common Algebraic Specification Language, for meta-notation. The modularity of this MSOS description was particularly useful during the redesign of Action Notation [18]. An MSOS of ML concurrency primitives has been provided [15] to facilitate a comparison of MSOS with SOS and evaluation-context (reduction) semantics for the same language. MSOS has not so far been used for giving a complete description of a major programming language, and it remains to be seen whether any pragmatic problems would arise when scaling up.

The continuing development of MSOS is motivated by the aim of optimizing the pragmatic aspects of the structural approach to operational semantics, partly for use when teaching semantics [19]. Several topics are in need of further work:

– For truly definitive descriptions of individual language constructs, a universal language-independent abstract syntax has to be established.
– A library of MSOS modules (with accompanying Prolog implementations) should be made available.
– Existing systems supporting animation and validation of SOS [5] might be adapted to support MSOS.
– A type-checker for MSOS should be provided.
– The study of bisimulation and other equivalences for MSOS, initiated in [13, 14], should be continued.

Readers who might be interested in contributing to the development of MSOS by working on these or other topics are requested to let the author know.

Acknowledgements The author is grateful to the organizers of AMAST 2002 for the invitation to talk on the topic of Modular SOS. Olivier Danvy and Jørgen Iversen suggested improvements to drafts of this paper; remaining infelicities are the sole responsibility of the author. Bartek Klin suggested the novel technique for dealing with exceptions. Leonardo de Moura suggested implementing MSOS directly in Prolog.

References

1. L. Aceto, W. J. Fokkink, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 1: Basic Theory, pages 197–292. Elsevier, 2001.
2. D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 119–129. ACM, 1992.
3. C. de O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rua Marquês de São Vicente 255, Gávea, Rio de Janeiro, RJ, Brazil, September 2001. `http://www.inf.puc-rio.br/~cbraga`.
4. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *AMAST 2000*, volume 1816 of *LNCS*, pages 407–421. Springer-Verlag, 2000.
5. P. H. Hartel. LETOS - a lightweight execution tool for operational semantics. *Software – Practice and Experience*, 29(15):1379–1416, Sept. 1999.

6. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, Mar. 2000.
7. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics.* Wiley, New York, 1990.
8. G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.
9. R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
10. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
11. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.
12. P. D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
13. P. D. Mosses. Foundations of modular SOS. Research Series RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/54`; full version of [14].
14. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available at `http://www.brics.dk/RS/99/54/`.
15. P. D. Mosses. A modular SOS for Action Notation. Research Series RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/56`. Full version of [16].
16. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at `http://www.brics.dk/RS/99/56/`.
17. P. D. Mosses. A modular SOS for ML concurrency primitives. Research Series RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/57/`.
18. P. D. Mosses. AN-2: Revised action notation—syntax and semantics. Available at `http://www.brics.dk/~pdm/papers/Mosses-AN-2-Semantics/`, Oct. 2000.
19. P. D. Mosses. Fundamental concepts and formal semantics of programming languages. Lecture Notes. Version 0.2, available from `http://www.brics.dk/~pdm/`, Sept. 2002.
20. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, Chichester, UK, 1992.
21. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
22. K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley, 1995.
23. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.