# Exploiting Labels in Structural Operational Semantics

Peter D. Mosses

BRICS Report Series RS-05-8

ISSN 0909-0878 February 2005

*Fundamenta Informaticae XX (2005) 1–15*
*IOS Press*

# Exploiting Labels in Structural Operational Semantics[*]

**Peter D. Mosses**
*BRICS[†]& Department of Computer Science, University of Aarhus*
*IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark*
*pdmosses@brics.dk*

**Abstract.** Structural Operational Semantics (SOS) allows transitions to be labelled. This is fully exploited in SOS descriptions of concurrent systems, but usually not at all in conventional descriptions of sequential programming languages.

This paper shows how the use of labels can provide significantly simpler and more modular descriptions of programming languages. However, the full power of labels is obtained only when the set of labels is made into a category, as in the recently-proposed MSOS variant of SOS.

**Keywords:** Structural operational semantics, SOS, modularity, MSOS, natural semantics

## 1. Introduction

Structural Operational Semantics (SOS) [24] is a well-known framework that can be used for specifying the semantics of concurrent systems [1, 11] and programming languages [12]. It has been widely taught, especially at the undergraduate level [6, 23, 24, 25, 27], and it is generally found to be significantly more accessible to students than denotational semantics.

In general, labels on transitions in SOS represent interaction possibilities, such as communication and/or synchronization between concurrent processes. In the usual interleaving SOS of CCS, for instance, labels are (atomic) actions equipped with a complementation operation, together with a special label $\tau$ that represents unobservable transitions [11]; for a non-interleaving SOS of CCS, labels can record also the locations of actions.

[*]Expanded version of a paper with the same title appearing in SAC'04, Proc. 2004 ACM Symposium on Applied Computing, ACM, 2004. http://doi.acm.org/10.1145/967900.968195.
[†]Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

When using SOS to define the semantics of sequential programming languages, however, labels are typically not used at all: all auxiliary information (such as environments and stores) is incorporated in the configurations of the transition system.

This paper shows how labels can be exploited in SOS descriptions of sequential languages to a much greater extent. As a somewhat unexpected bonus, we shall see how labels allow us to give a big-step SOS for constructs that involve interleaving – something that was commonly believed to be impossible. The presented techniques for using labels were all developed in connection with a modular variant, MSOS [16, 20, 21], of the conventional SOS framework. However, improved modularity is not the only benefit of the techniques, which can also be exploited when modularity is of no concern.

Compared with SOS, MSOS goes to the opposite extreme regarding labels when describing sequential programming languages: they are exploited as much as possible. Configurations in MSOS are simply abstract syntax trees (together with computed values), so the labels on transitions have to incorporate *all* auxiliary entities.

Taking environments and stores out of configurations and putting them in labels gives a clear separation between syntactic entities and those representing semantic information: configurations in MSOS always represent what remains to be computed (as usual in the SOS of concurrent systems), and the label on a transition represents all the "information processing" associated with it: the information available for inspection, any updates to that information, and any new information produced by the transition itself.

The information processing of transitions in a computation is subject to the obvious constraint that the part of it available for inspection remains stable, except when updated by the transitions themselves. This constraint is represented in MSOS by taking the labels to be the arrows of a category, and requiring labels on adjacent transitions to be composable. The objects of the label category correspond to states of the processed information. Identity arrows are naturally used to label unobservable (silent) transitions.

It appears that this way of combining the familiar notions of labelled transition system and category is novel, and has not previously been exploited in connection with operational semantics of programming languages and concurrent systems.

The rest of the paper is organized as follows: Section 2 recalls the definition of MSOS from [16, 20], pointing out the differences from the conventional SOS framework. Sections 3, 4, and 5 give simplified examples of MSOS, illustrating how labels are used. Section 6 explains how these simple examples can be combined, and how a high degree of modularity can be obtained. Section 7 shows how interleaving can be described in a variant of so-called big-step SOS (Natural Semantics). Section 8 discusses bisimulation equivalence in MSOS. Section 9 explains the relationship of the work presented here to previous work.

## 2. Modular SOS

The conventional SOS framework is based on labelled transition systems (LTS):

**Definition 2.1.** A *labelled transition system* $LTS$ is a quadruple $\langle \Gamma, A, \longrightarrow, T\rangle$ consisting of a set $\Gamma$ of configurations $\gamma$, a set $A$ of labels $\alpha$, a ternary relation $\longrightarrow\ \subseteq \Gamma \times A \times \Gamma$ of labelled transitions ($\langle \gamma, \alpha, \gamma'\rangle \in\ \longrightarrow$ is written $\gamma \xrightarrow{\alpha} \gamma'$), and a set $T \subseteq \Gamma$ of terminal configurations, such that $\gamma \xrightarrow{\alpha} \gamma'$ implies $\gamma \notin T$.

A *computation* in an $LTS$ (from $\gamma_0$) is a finite or infinite sequence of successive transitions $\gamma_i \xrightarrow{\alpha_i} \gamma_{i+1}$ (written $\gamma_0 \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \cdots$), such that when the sequence terminates with $\gamma_n$ we have $\gamma_n \in T$.

The *trace* of an infinite computation $\gamma_0 \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \cdots$ is the sequence $\alpha_1\alpha_2\ldots$; the trace of a finite computation $\gamma_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \gamma_n$ is the sequence $\alpha_1 \ldots \alpha_n\gamma_n$.

There are no restrictions on the set of configurations $\Gamma$ of an LTS: the (abstract) syntax of programs and their parts (expressions, declarations, commands, etc.) and the values that they compute are allowed as components of configurations $\gamma \in \Gamma$, but there may also be further components. The set of labels $A$ of an LTS is entirely unconstrained. A computation of an LTS is either infinite, or terminates in a distinguished set $T \subseteq \Gamma$ of final configurations.

In the MSOS variant of SOS, configurations are constrained, the set of labels is given some structure, and computations are required to respect that structure. The following kind of generalized transition system was introduced in [16]:[1]

**Definition 2.2.** A *generalized transition system* $GTS$ is a quadruple $\langle \Gamma, \mathbb{A}, \longrightarrow, T\rangle$ where $\mathbb{A}$ is a category with morphisms $A$, such that $\langle \Gamma, A, \longrightarrow, T\rangle$ is a labelled terminal transition system LTS.

A *computation* in a $GTS$ is a computation in the underlying $LTS$ such that its trace is a path in the category $\mathbb{A}$: whenever a transition labelled $\alpha$ is followed immediately by a transition labelled $\alpha'$, the labels $\alpha, \alpha'$ are required to be composable in $\mathbb{A}$.

MSOS does not allow auxiliary information in configurations. Thus the initial configuration of a computation is always just an abstract syntax tree, and the final configuration (if any) is just a computed value. (As in SOS, intermediate configurations generally involve mixtures of abstract syntax and computed values, where some nodes have been replaced by the values that they have already computed.)

In MSOS, the set of labels $\alpha \in A$ of an LTS is made into the set of *arrows* of a category $\mathbb{A}$ by equipping it with a partial composition, written $\alpha_1 \mathbin{;} \alpha_2$, and by distinguishing a set of *identity labels* $\mathbb{I} \subseteq \mathbb{A}$, satisfying the usual axioms: composition is associative (when defined) and identities are left and right units. The elements of $\mathbb{I}$ are taken as the *objects* of the category $\mathbb{A}$. The set of arrows between two objects $\alpha_1, \alpha_2 \in \mathbb{I}$ consists of all $\alpha$ such that both $\alpha_1 \mathbin{;} \alpha$ and $\alpha \mathbin{;} \alpha_2$ are defined.

In MSOS, computations are restricted to those whose labels trace (possibly infinite) paths through the label category: when a transition labelled $\alpha_1$ is followed immediately by a transition labelled $\alpha_2$, the composition $\alpha_1 \mathbin{;} \alpha_2$ must be defined.

In some other approaches to operational semantics, transitions between configurations, rather than labels on transitions, are taken as the arrows of a category. This requires the transition relation to be both transitive and reflexive, neither of which is appropriate for MSOS. (Transitivity would make it problematic to describe constructs which rely on atomicity, such as test-and-set, and reflexivity would imply that computations could always be continued with 'stuttering' transitions.)

**Proposition 2.1.** For each GTS $\langle \Gamma, \mathbb{A}, \longrightarrow, T\rangle$, an LTS $\langle \Gamma^\bullet, A^\bullet, \longrightarrow^\bullet, T^\bullet\rangle$ can be constructed such that for each computation of the GTS, there is a computation of the LTS with the same trace, and vice versa.

This result is proved in [21]. The construction is straightforward: each configuration of the LTS is a pair consisting of a configuration of the LTS and an object of the label category, and the transition relation is defined accordingly.

[1]Generalized transition systems were called "arrow-labelled" in [16].

## 3. Environments

The current bindings of identifiers during a computation are usually represented by environments $\rho \in \mathit{Env}$, where $\mathit{Env}$ is a set of finite maps.

In SOS, environments are usually taken as components of configurations, e.g., $\Gamma = \cdots \times \mathit{Env}$, so transitions might go from $\langle \ldots, \rho \rangle$ to $\langle \ldots, \rho' \rangle$. In practice, transitions never change the environment, and the notation $\rho \vdash \langle \ldots \rangle \xrightarrow{\alpha} \langle \ldots \rangle$ is usually introduced to abbreviate $\langle \ldots, \rho \rangle \xrightarrow{\alpha} \langle \ldots, \rho \rangle$.

An alternative to taking environments as components of configurations in SOS is to incorporate them in *labels*. Here is an example, illustrating how a (small-step) SOS for some simple expressions would look when written this way:

| | | | |
|---|---|---|---|
| Numbers | $n \in \mathbb{N}$ | $=$ | $\{0, 1, 2, \ldots\}$ |
| Truth-values | $t \in \mathbb{T}$ | $=$ | $\{\mathbf{tt}, \mathbf{ff}\}$ |
| Identifiers | $x \in \mathit{Id}$ | $=$ | $\{\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \ldots\}$ |
| Binary Ops. | $\mathit{bop} \in \mathit{Bop}$ | $=$ | $\{+, -, *, \ldots, <, =\}$ |
| Constants | $\mathit{con} \in \mathit{Con}$ | $::=$ | $t \mid n$ |
| Expressions | $e \in \mathit{Exp}$ | $::=$ | $\mathit{con} \mid e_0\ \mathit{bop}\ e_1 \mid$ |
| | | | $x \mid \mathbf{let}\, x = e_0\, \mathbf{in}\, e_1 \mid \ldots$ |
| Computed Values | $T$ | $=$ | $\mathbb{N} \cup \mathbb{T}$ |
| Bound Values | $\mathit{BV}$ | $=$ | $\mathbb{N} \cup \mathbb{T}$ |
| Environments | $\rho \in \mathit{Env}$ | $=$ | $\mathit{Id} \to_{\mathrm{fin}} \mathit{BV}$ |
| Labels | $A$ | $=$ | $\mathit{Env}$ |

$$\frac{e_0 \xrightarrow{\rho} e_0'}{e_0\ \mathit{bop}\ e_1 \xrightarrow{\rho} e_0'\ \mathit{bop}\ e_1} \tag{1}$$

$$\frac{e_1 \xrightarrow{\rho} e_1'}{e_0\ \mathit{bop}\ e_1 \xrightarrow{\rho} e_0\ \mathit{bop}\ e_1'} \tag{2}$$

$$\frac{\mathit{bop} = +, \quad n = n_0 + n_1}{n_0\ \mathit{bop}\ n_1 \xrightarrow{\rho} n} \tag{3}$$

$$\frac{\rho(x) = \mathit{con}}{x \xrightarrow{\rho} \mathit{con}} \tag{4}$$

$$\frac{e_0 \xrightarrow{\rho} e_0'}{\mathbf{let}\, x = e_0\, \mathbf{in}\, e_1 \xrightarrow{\rho} \mathbf{let}\, x = e_0'\, \mathbf{in}\, e_1} \tag{5}$$

$$\frac{e_1 \xrightarrow{\rho[x = \mathit{con}_0]} e_1'}{\mathbf{let}\, x = \mathit{con}_0\, \mathbf{in}\, e_1 \xrightarrow{\rho} \mathbf{let}\, x = \mathit{con}_0\, \mathbf{in}\, e_1'} \tag{6}$$

$$\mathbf{let}\, x = \mathit{con}_0\, \mathbf{in}\, \mathit{con}_1 \xrightarrow{\rho} \mathit{con}_1 \tag{7}$$

Unfortunately, the above example does *not* give the intended operational semantics in conventional SOS, since the environments used as labels on successive transitions are unconstrained. In MSOS, however, we can accurately reflect that the environment does not change from one transition to the next by taking the label category for environments to be *discrete*: all arrows of the category are identities, and the composition $\rho\,;\rho'$ is defined only when $\rho = \rho'$.

## 4. Stores

The values currently stored at memory locations during a computation are represented by abstract stores $\sigma \in S$, where $S$ is a set of finite maps.

In SOS, stores are usually taken as components of configurations, e.g., $\Gamma = \cdots \times S$, so a transition goes from $\langle \ldots, \sigma \rangle$ to $\langle \ldots, \sigma' \rangle$, where the difference between $\sigma$ and $\sigma'$ corresponds to the (observable) updates made by the transition. For constructs that can inspect but not update the stored information, $\sigma = \sigma'$.

An alternative to taking stores as components of configurations is to incorporate *pairs* of stores $(\sigma, \sigma')$ in labels. The store $\sigma$ represents information that can be inspected at the beginning of the transition, and $\sigma'$ the possibly-updated information left at the end of the transition. Let us see how a small-step SOS for some familiar commands would look when written this way. (For simplicity, we disregard environments here, and for uniformity, we assume that expressions could have side-effects when evaluated.)

$$\begin{array}{llcl}
\text{Locations} & l \in L & = & \{\mathbf{l_0}, \mathbf{l_1}, \mathbf{l_2}, \ldots\} \\
\text{Expressions} & e \in \mathit{Exp} & ::= & l \mid \mathit{con} \\
\text{Commands} & c \in \mathit{Com} & ::= & \mathbf{nil} \mid l := e \mid c_0\,; c_1 \mid \\
 & & & \mathbf{if}\, e\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \mid \ldots \\
\text{Computed Values} & T & = & \mathbb{N} \cup \mathbb{T} \cup \{\mathbf{nil}\} \\
\text{Stored Values} & SV & = & \mathbb{N} \\
\text{Stores} & \sigma \in S & = & L \to_{\text{fin}} SV \\
\text{Labels} & A & = & S \times S
\end{array}$$

$$\frac{\sigma(l) = n}{l \xrightarrow{(\sigma,\sigma)} n} \tag{8}$$

$$\frac{e \xrightarrow{(\sigma,\sigma')} e'}{l := e \xrightarrow{(\sigma,\sigma')} l := e'} \tag{9}$$

$$l := n \xrightarrow{(\sigma,\sigma[l=n])} \mathbf{nil} \tag{10}$$

$$\frac{c_0 \xrightarrow{(\sigma,\sigma')} c_0'}{c_0\,; c_1 \xrightarrow{(\sigma,\sigma')} c_0'\,; c_1} \tag{11}$$

$$\mathbf{nil}\,; c_1 \xrightarrow{(\sigma,\sigma)} c_1 \tag{12}$$

$$\frac{e \xrightarrow{(\sigma,\sigma')} e'}{\mathbf{if}\,e\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1 \xrightarrow{(\sigma,\sigma')} \mathbf{if}\,e'\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1} \tag{13}$$

$$\mathbf{if}\,\mathbf{tt}\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1 \xrightarrow{(\sigma,\sigma)} c_0 \tag{14}$$

$$\mathbf{if}\,\mathbf{ff}\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1 \xrightarrow{(\sigma,\sigma)} c_1 \tag{15}$$

Obviously, the above example does *not* give the intended operational semantics in conventional SOS, since the pairs of stores used as labels on successive transitions are unconstrained, whereas we should require that the $\sigma'$ at the end of each transition in a computation is identical to the $\sigma$ at the beginning of the following transition – assuming that the stored information does not change spontaneously between transitions.

In MSOS, we can accurately reflect the above requirement by making $S \times S$ into a *preorder* category in the obvious way: the identities are of the form $(\sigma, \sigma)$, corresponding to single stores, and composition is defined by $(\sigma, \sigma')\;;\,(\sigma', \sigma'') = (\sigma, \sigma'')$, otherwise undefined.

## 5. Abrupt Termination

Abrupt termination occurs when the execution of the program reaches a construct that requires the normal flow of control to be abandoned. Such constructs include commands for breaking out of the (smallest) enclosing loop, returning from a procedure activation, jumping to a label, raising an exception, or simply stopping the program prematurely. Other constructs allow particular forms of abnormal termination to be detected and handled, whereupon the normal flow of control may be continued. All remaining constructs are said to *propagate* abrupt termination, terminating abruptly whenever one of their components does.

In SOS, abrupt termination of a part of the program is usually represented by its computation reaching some abnormal final configuration. Rules for constructs that detect and handle abrupt termination distinguish abnormal final configurations from those that can arise due to normal termination. A rule is needed for each possibility of propagation of abrupt termination; unfortunately, such rules can be quite numerous, and rather tedious to specify.

The following technique was discovered by B. Klin (BRICS, Univ. of Aarhus) in connection with MSOS, but it is directly applicable also in conventional SOS.

An alternative to the conventional use of final configurations to distinguish between normal and abrupt termination is to let the *label* on each transition indicate whether or not abrupt termination is required. Such labels can be propagated *uniformly* to the enclosing constructs, thus avoiding the need for tedious extra rules.

To illustrate this novel technique, let us take a minimalistic example involving a stop-command which, when executed, is supposed to stop the execution of the enclosing program immediately. We introduce a construct '$\mathbf{program}\,c$' indicating the top of the entire program, and a corresponding final configuration for program computations, '$\mathbf{end}$'. (The description of breaks, returns, and exceptions would be quite similar; that of jumps to labels is complicated only in respect of specifying the binding of labels to “local continuations”.) For simplicity, let us disregard environments and stores here.

| | | | |
|---|---|---|---|
| Commands | $c \in \mathit{Com}$ | $::=$ | $\mathbf{nil} \mid c_0 \,;\, c_1 \mid \mathbf{stop} \mid \ldots$ |
| Programs | $p \in \mathit{Prog}$ | $::=$ | $\mathbf{program}\, c \mid \mathbf{end}$ |
| Computed Values | $T$ | $=$ | $\{\mathbf{nil}, \mathbf{end}\}$ |
| Labels | $\alpha \in A$ | $=$ | $\{\mathit{stopped}, (\,)\}$ |

$$\frac{c_0 \xrightarrow{\alpha} c_0'}{c_0 \,;\, c_1 \xrightarrow{\alpha} c_0' \,;\, c_1} \tag{16}$$

$$\mathbf{nil} \,;\, c_1 \xrightarrow{(\,)} c_1 \tag{17}$$

$$\mathbf{stop} \xrightarrow{\mathit{stopped}} c \tag{18}$$

$$\frac{c \xrightarrow{(\,)} c'}{\mathbf{program}\, c \xrightarrow{(\,)} \mathbf{program}\, c'} \tag{19}$$

$$\frac{c \xrightarrow{\mathit{stopped}} c'}{\mathbf{program}\, c \xrightarrow{(\,)} \mathbf{end}} \tag{20}$$

$$\mathbf{program\,nil} \xrightarrow{(\,)} \mathbf{end} \tag{21}$$

The above example gives the intended semantics both in conventional SOS and in MSOS, since no constraint on the labels of successive transitions is needed. The semantics specified for $\mathbf{stop}$ is analogous to that of a signalling action in a concurrent system; that for $\mathbf{program}\, c$ corresponds to a kind of *synchronization* between the entire program and the first executed occurrence of $\mathbf{stop}$ in its body. Notice that the label carries the abrupt termination signal upwards through the derivation of a single transition.

In MSOS, we represent the lack of constraints on adjacent labels by making the set of labels into a *monoid*: considered as a category, a monoid has a single object, hence its composition is total. For the above example, it is enough to take a two-element monoid with unit $(\,)$ and non-unit $\mathit{stopped}$. (The MSOS of interactive input and output requires label categories to be free monoids; these could be used here too, although sequences with more than one ‘$\mathit{stopped}$’ would never arise in the specified computations.)

## 6. Combinations

The examples given in the preceding sections have illustrated the use of various techniques separately, for simplicity. But what if we want to use all the proposed techniques at once, in the same description? This requires labels that can incorporate several components.

For combining abstract syntax with environments and stores in SOS, Cartesian products are used. One can do the same in SOS for combining separate components of labels, for instance taking $A$ to be $\mathit{Env} \times S \times S \times \{(\,), \mathit{stopped}\}$, and writing labels in rules as $\langle \rho, \sigma, \sigma', \alpha \rangle$.

Recall that in MSOS, we use different kinds of label category for environments, stores, and abrupt termination. The obvious way of combining these separate label categories into a single category is to form a product category, where composition is defined component-wise: the composition of the tuples is defined if and only if the composition of each corresponding pair of components is defined.

It is possible to construct label categories incrementally (starting from the trivial category) using label category transformers [16], which are loosely analogous to simple monad transformers [8, 13]. An alternative is to specify the desired product category directly, given notation for the three basic ways of constructing the component label categories from sets, e.g.:

$$\mathbb{A} = \mathbf{Discrete}(\mathit{Env}) \times \mathbf{Pair}(S) \times \{\mathit{stopped}\}^?$$

It is clearly desirable to avoid the need for large-scale reformulation of rules when adding new, unforeseen components to labels: e.g., when adding exceptions to a language, we would not want to have to add explicit propagation of the corresponding label component to all the existing rules.

In fact, rules for constructs concerned only with normal flow of control (sequencing, conditionals, iterations, etc.) are naturally formulated without concern for components of labels. Let the meta-variable $X$ range over arbitrary arrows in $\mathbb{A}$, and $U$ over the subset $\mathbb{I}$ of identity arrows. Then the following rules for sequential commands subsume those previously given in both Sects. 4 and 5:

$$\frac{c_0 \xrightarrow{X} c_0'}{c_0\,;c_1 \xrightarrow{X} c_0'\,;c_1} \tag{22}$$

$$\mathbf{nil}\,;c_1 \xrightarrow{U} c_1 \tag{23}$$

In (22) above, the use of $X$ both in the premise and in the conclusion ensures that the transitions for $c_0$ and for '$c_0\,;c_1$' have the same label, whatever components that label might have. The restriction of the label $U$ to an identity arrow in (23) means that the transition for '$\mathbf{nil}\,;c_1$' must leave any store component unchanged, and any component used for indicating abrupt termination must be void ( ), as expected for a transition that should be completely unobservable. If labels have an environment component, (22) also requires $c_0$ have the same environment as '$c_0\,;c_1$'.

Some rules involve setting or inspecting particular components of labels. Given the definition of the label category $\mathbb{A}$, tuple patterns could be used for this, e.g.:

$$\frac{e_1 \xrightarrow{\langle\rho[x=\mathit{con}_0],(\sigma,\sigma'),\alpha\rangle} e_1'}{\mathbf{let}\,x = \mathit{con}_0\,\mathbf{in}\,e_1 \xrightarrow{\langle\rho,(\sigma,\sigma'),\alpha\rangle} \mathbf{let}\,x = \mathit{con}_0\,\mathbf{in}\,e_1'} \tag{24}$$

However, this notation is quite tedious when there are many components, but only one or two of them are relevant the rule concerned. Moreover, adding a further component would require its insertion in all rules that use the tuple patterns.

Another possibility is to define *set* and *get* operations on $\mathbb{A}$, with symbolic indices as arguments, as in [16]. This corresponds closely to using an *indexed* product category, where the components are unordered. The arrows of such a category correspond to records in programming languages. ML provides a suggestive notation for record patterns, using '$\ldots$' as a variable to stand for any number of indexed components; this allows (24) above to be written as follows:

$$\frac{e_1 \xrightarrow{\{\rho=\rho_0[x=con_0],\ldots\}} e_1'}{\mathbf{let}\, x = con_0 \,\mathbf{in}\, e_1 \xrightarrow{\{\rho=\rho_0,\ldots\}} \mathbf{let}\, x = con_0 \,\mathbf{in}\, e_1'} \tag{25}$$

Here, in contrast to in ML, different occurrences of '. . . ' in the same rule stand for the *same* set of record components. Restrictions of (parts of) labels to identity arrows need to be stated as premises, e.g.:

$$\frac{\rho_0(x) = con, \quad U = \{\rho=\rho_0,\ldots\}}{x \xrightarrow{U} con} \tag{26}$$

$$\frac{\sigma_0(l) = n, \quad U = \{\sigma=\sigma_0,\ldots\}}{l \xrightarrow{U} n} \tag{27}$$

$$\frac{U = \{\sigma=\sigma_0, \sigma'=\sigma_0,\ldots\}}{l := n \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[l=n],\ldots\}} \mathbf{nil}} \tag{28}$$

$$\frac{U = \{\alpha'=(\,),\ldots\}}{\mathbf{stop} \xrightarrow{\{\alpha'=stopped,\ldots\}} c} \tag{29}$$

The remaining rules that need to refer to particular components of labels do not require any extra premises:

$$\frac{c \xrightarrow{\{\alpha'=(\,),\ldots\}} c'}{\mathbf{program}\, c \xrightarrow{\{\alpha'=(\,),\ldots\}} \mathbf{program}\, c'} \tag{30}$$

$$\frac{c \xrightarrow{\{\alpha'=stopped,\ldots\}} c'}{\mathbf{program}\, c \xrightarrow{\{\alpha'=(\,),\ldots\}} \mathbf{end}} \tag{31}$$

Note the systematic use of primes on the indexes of the records: an index that occurs only unprimed (such as $\rho$ above) refers to a component from a discrete category; one which occurs both unprimed and primed (such as $\sigma$) refers to the first, resp. second, component of a pair coming from a preorder category; and an index which occurs only primed (such as $\alpha$) refers to a component from a monoid category. An unprimed index thus always refers to information available at the *beginning* of a transition, and a primed index refers to information determined at the *end* of a transition.

In MSOS, the description of each programming construct can often be given definitively, once and for all. The degree of modularity can be so high that the rules given for each construct should *never* require reformulation when other constructs are added to (or removed from) the described language. This comes entirely from the use of label categories together with notation allowing particular components of labels to be set or inspected independently of the presence of other components (such as that provided together with label category transformers [16], or the record notation illustrated above, which was borrowed from ML).

The novel technique described in Sect. 5 avoids the need for a lot of extra rules when adding constructs that involve abrupt termination.

The notational overhead of MSOS compared to conventional SOS is quite minor (in fact the MSOS rules for many constructs are *more* concise than the corresponding SOS rules) and is in any case completely outweighed by the possibility of reusing MSOS rules in the semantic descriptions of many different programming languages.

## 7. Modular Natural Semantics

All the illustrations of MSOS given above have been formulated in the so-called *small-step* style, where a computation is a sequence of transitions from an initial configuration through intermediate configurations, leading either to a final configuration or to nontermination. For sequential programs, each transition generally involves a *single* construct, such as the application of an arithmetic operation or an assignment command, where at least some of its components have already been replaced by their computed values. Only for programs involving synchronization does a transition involve two (or more) constructs in different parts of the program.

In fact MSOS can be used for the big-step style of SOS too. This style of SOS is also known as Natural Semantics [7], and it was used for the definition of Standard ML [12].

Like the small-step style of SOS, Natural Semantics (NS) uses rules to specify computations, and auxiliary entities such as environments and stores to represent the information processed by computations. However, it doesn't involve *sequences* of transitions at all: a computation goes straight from the syntax of a construct to its computed value – depending on computations for some or all the components of the construct. Nonterminating computations are ignored altogether, since they would require infinitely-deep derivations.

The evaluation relation of an NS can be regarded as a degenerate transition relation where there are no intermediate configurations. The same label categories that were used for environments and stores in MSOS can be used to obtain a modular variant, say MNS, of NS. Now, however, the rule for the evaluation of a construct often involves the evaluation of several components, and the labels used for these sub-evaluations have to be composed. Here is an example:

$$\frac{c_0 \xrightarrow{X_0} \mathbf{nil}, \quad c_1 \xrightarrow{X_1} \mathbf{nil}, \quad X = X_0 \,;\, X_1}{c_0 \,;\, c_1 \xrightarrow{X} \mathbf{nil}} \tag{32}$$

Notice that the explicit composition of $X_0$ and $X_1$ above makes it clear that the intended order of computation of $c_0$ and $c_1$ is from left to right – independently of the order in which the premises of the rule are written. Similarly in the following rule for evaluation of addition expressions:

$$\frac{e_0 \xrightarrow{X_0} n_0, \quad e_1 \xrightarrow{X_1} n_1, \quad X = X_0 \,;\, X_1}{e_0 + e_1 \xrightarrow{X} n_0 + n_1} \tag{33}$$

Although one can easily specify that expression evaluation is a nondeterministic choice between left-to-right and right-to-left (by adding a rule identical to (33) except for specifying $X = X_1 \,;\, X_0$), it is not so straightforward to specify that arbitrary interleaving of sub-expression evaluations is allowed. The difficulty of specifying interleaving in conventional NS is generally regarded as motivation for using the small-step style of SOS when describing languages where any construct involves interleaving. In MNS, however, we may exploit an unorthodox label category where (in essence) we take the monoid category $(S \times S)^*$ (instead of the usual $\mathbf{Pair}(S \times S)$) as the component for stores. The computation of an atomic construct always gives a single pair of stores, but sequencing of constructs gives rise to longer sequences of pairs. The point of this is that now we can define also an operation $\mathit{interleavings}(X_0, X_1)$ on labels that produces the set of labels with all possible interleavings of the sequences of pairs of stores in $X_0$ and $X_1$. A rule for interleaved expression evaluation is then specified as follows:

$$\frac{e_0 \xrightarrow{X_0} n_0,\ e_1 \xrightarrow{X_1} n_1,\ X \in \mathit{interleavings}(X_0, X_1)}{e_0 + e_1 \xrightarrow{X} n_0 + n_1} \tag{34}$$

The lack of insistence on composability (in the usual sense) of the pairs of stores in these sequences reflects that interleaving may cause a completely arbitrary update to the store between any two transitions. When a construct (such as an entire program) is protected from external interleaving, all that is needed is to select those labels with sequences that can be composed to give a single pair of stores.

Unfortunately, the discovery of how to specify interleaving in MNS remedies only one of several drawbacks of the big-step style in relation to the small-step style of MSOS: apart from the lack of reflection of nonterminating computations in MNS, the novel technique illustrated for abrupt termination in Sect. 5 is not applicable in MNS. (To see this, consider $\mathbf{program}\, c$ where $c$ is $\mathbf{stop}\,; c_1$ and $c_1$ is any nonterminating command. Since there is no $X_1$ such that $c_1 \xrightarrow{X_1} \mathbf{nil}$, rule (32) cannot be used to derive any transition at all for $c$. Thus no big-step rules for $\mathbf{program}\, c$ would be able to distinguish between $c$ and $c_1$. This implies that the $\mathbf{stop}$ command does not stop the program.)

Thus the recommendation is to use small-step (M)SOS for constructs whenever their operational semantics involves (or might later be extended to involve) either nontermination or abrupt termination. For constructs that *inherently* involve only normal termination (e.g., decimal notation for unbounded integers, and type expressions), the order of evaluation of component constructs is often irrelevant, and MNS may then be preferable to MSOS.

## 8. Equivalence in MSOS

The development of MSOS has so far been focussed on establishing appropriate foundations for modular specifications of programming languages, and on developing an appropriate meta-notation for writing such specifications. The study of equivalences based on MSOS is still at an early stage. Although the standard definitions carry straight over from SOS to MSOS, and allow proofs of general algebraic properties, it is questionable whether the resulting equivalences are large enough to allow reasoning about the MSOS of specific programs.

### 8.1. Strong Bisimulation

An MSOS defines a generalized transition system $\mathit{GTS} = \langle \Gamma, \mathbb{A}, \longrightarrow, T \rangle$ with an underlying labelled transition system $\mathit{LTS} = \langle \Gamma, A, \longrightarrow, T \rangle$, where $A$ is the set of morphisms of the label category $\mathbb{A}$. Adjacent labels in computations are required to be composable in $\mathbb{A}$. Let us first recall the usual notion of strong bisimulation for ordinary labelled transition systems [10], adjusted to take account of terminal configurations:

**Definition 8.1.** Let $LTS = \langle \Gamma, A, \longrightarrow, T \rangle$ be a labelled transition system. $R \subseteq \Gamma \times \Gamma$ is a *strong bisimulation* iff $\langle \gamma_1, \gamma_2 \rangle \in R$ implies, for all $\alpha \in A$,

- whenever $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ then for some $\gamma_2'$, $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ and $\langle \gamma_1', \gamma_2' \rangle \in R$;
- whenever $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ then for some $\gamma_1'$, $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ and $\langle \gamma_1', \gamma_2' \rangle \in R$; and
- whenever $\gamma_1 \in T$ or $\gamma_2 \in T$ then $\gamma_1 = \gamma_2$.

$\gamma_1$, $\gamma_2$ are *strongly bisimilar*, written $\gamma_1 \sim \gamma_2$, iff $\langle \gamma_1, \gamma_2 \rangle \in R$ for some strong bisimulation $R$.

The above definition of strong bisimulation carries over unchanged from LTS to GTS, and the usual proof techniques are available. Since the configurations $\gamma$ of the GTS defined by an MSOS are purely syntax and computed values, we obtain bisimulation and bisimilarity relations on programs (and parts of programs) without the need to quantify explicitly over auxiliary entities such as environments and stores. In fact an MSOS for a programming language resembles an SOS for a process algebra, the main difference being in the nature of the labels.

This straightforward definition of strong bisimulation for GTS is insensitive to whether adjacent labels in computations are composable or not, since for each pair of configurations, we consider all possible labels on their next transitions, without regard to the labels on the transitions that led to those configurations. For general algebraic properties (e.g. commutativity, associativity) such insensitivity clearly does not matter: one has to prove that syntactically-distinct programs do in fact have the same possibilities for the flow of control between their unknown parts, regardless of the information which is processed by those parts.

Suppose, however, that we are to prove equivalence of programs involving specific bindings of identifiers to values, or specific assignments of values to variables, where the combination of the syntactic configuration and the auxiliary information carried by the labels can determine the future flow of control. In this case, the relevant point is that the labels on transitions reveal *all* components of the information being processed: two programs can only be in a bisimulation when they start from the same environment, and make exactly matching changes to the store at each transition. The fact that stores are included in labels ensures that bisimilar programs always have the same store at each transition.

The original definition of strong bisimulation for MSOS [16] was based on the reduction from GTS to LTS, and involved binary relations between pairs consisting of GTS configurations and objects of the label category. It now appears that it was unnecessarily complicated.

A full treatment should take account of the fact that environments in practice often have syntactic components, for instance closures representing functions with static scopes for bindings. Since environments occur as components of labels in MSOS, it's too restrictive to insist on labels being *identical* in connection with bisimulation: their syntactic components should be allowed be in the bisimulation relation themselves. The same goes for the computed values, which may also have syntactic components. Thus a *higher-order* bisimulation is needed, similar to that defined for use with higher-order process algebra where processes can be passed as values. (There has as yet been no experience of using higher-order bisimulation to prove properties of languages specified in MSOS, so we omit the definition here.)

## 8.2. Weak Bisimulation

An MSOS for a programming language involves many unobservable transitions, for instance arising due to applying arithmetic operations to the values of sub-expressions. Sometimes, one can avoid unobservable transitions by taking account of the case when a component construct is making a transition to a final state, as in the SOS rules for command sequencing in Plotkin's notes [24], but it is not clear that the extra bother of doing that is worthwhile. For a general notion of equivalence, it is desirable to allow (finite sequences of) unobservable transitions to be ignored.

In studies of process algebra, many variations on the theme of weak bisimulation have been defined, based on the assumption that unobservable transitions are always being labelled with a special silent action, conventionally written $\tau$. In MSOS, we generally have a large set of labels for unobservable transitions: all the identity morphisms of the label category $\mathbb{A}$, so we do not need to add $\tau$ to our labels. Moreover, definitions of weak bisimulation don't depend on $\tau$ being a constant (we could regard it formally as a meta-variable ranging over the set of identity morphisms).

Thus the standard definition of weak bisimulation [10] is formulated for MSOS as follows (branching and other varieties of bisimulation would be defined analogously):

**Definition 8.2.** Let $\langle \Gamma, \mathbb{A}, \longrightarrow, T \rangle$ be a generalized transition system, and $A$ the set of morphisms of the category $\mathbb{A}$. $R \subseteq \Gamma \times \Gamma$ is a *weak bisimulation* iff $\langle \gamma_1, \gamma_2 \rangle \in R$ implies, for all $\alpha \in A$,

- whenever $\gamma_1 \overset{\alpha}{\Longrightarrow} \gamma_1'$ then for some $\gamma_2'$, $\gamma_2 \overset{\hat{\alpha}}{\Longrightarrow} \gamma_2'$ and $\langle \gamma_1', \gamma_2' \rangle \in R$;
- whenever $\gamma_2 \overset{\alpha}{\Longrightarrow} \gamma_2'$ then for some $\gamma_1'$, $\gamma_1 \overset{\hat{\alpha}}{\Longrightarrow} \gamma_1'$ and $\langle \gamma_1', \gamma_2' \rangle \in R$; and
- whenever $\gamma_1 \in T$ or $\gamma_2 \in T$ then $\gamma_1 = \gamma_2$.

where:

- $\overset{\alpha}{\Longrightarrow}$ is defined as the composition $\longrightarrow^* \xrightarrow{\alpha} \longrightarrow^*$,
- $\overset{\hat{\alpha}}{\Longrightarrow}$ is defined as $\longrightarrow^*$ when $\alpha$ is an identity morphism, otherwise as $\overset{\alpha}{\Longrightarrow}$,
- $\longrightarrow$ is the union of $\xrightarrow{\alpha'}$ for all identity morphisms $\alpha'$, and
- $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$.

## 9. Related Work

The present paper shows how MSOS arises as a natural consequence of greater exploitation of labels in SOS descriptions of programming languages. The original presentation of MSOS [15, 16] focusses on foundational aspects, such as the generalization of labelled transition systems and bisimulation to categories of labels; it also demonstrates that MSOS rules for some pure functional programming language constructs do not require any reformulation when expressions are enriched so as to allow side-effects and/or concurrency. A full case study [19] on the use of MSOS to describe the core of Concurrent ML provides a basis for comparison between an MSOS and a reduction semantics for the same language. A recent presentation of MSOS [20] addresses pragmatic aspects: it introduces a more perspicuous notation for labels (the notation used in the present paper incorporates further refinements, notably the use of ML's record patterns), it explains how MSOS descriptions can be transcribed into (equally-modular) interpreters in Prolog, and it gives a wide range of illustrative examples of MSOS rules, including those for ML-style exceptions.

The development of MSOS was stimulated by the need for a modular definition of the action notation used in action semantics [14, 22, 26] (see [5] for a more recent presentation). The original definition of action notation was given using SOS [14]; the redefinition in MSOS is reported in [17]. Braga [2] investigates the possibility of prototyping languages according to their action semantics via an implementation of MSOS in Maude; see also [3, 4]. Braga and Meseguer [9] propose an alternative modular form of SOS, and explain its relationship to MSOS.

## 10. Conclusion

The novel techniques presented here show how labels can be exploited to a much greater extent than usual in the operational semantics of programming languages. The incorporation of environments and stores in labels requires the use of the label categories provided by MSOS, but the novel treatment of abrupt termination is applicable also in conventional SOS.

Despite the somewhat surprising possibility of giving a big-step MSOS for interleaving constructs, the small-step style is still recommended for all constructs whose semantics might involve abrupt termination or nontermination.

**Acknowledgements** Thanks to the organizing committee of CS&P'03 for the invitation to give a talk at the meeting, and in particular to Ludwik Czaja for encouragement and patience in connection with the preparation of this paper. The author is supported by BRICS (Basic Research in Computer Science), funded by the Danish National Research Foundation.

## References

[1] Aceto, L., Fokkink, W., Verhoef, C.: Structural Operational Semantics, in: *Handbook of Process Algebra* (J. A. Bergstra, A. Ponse, S. A. Smolka, Eds.), chapter 3, Elsevier Science, 2001, 197–292.

[2] de O. Braga, C.: *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*, Ph.D. Thesis, Pontifícia Universidade Católica do Rio de Janeiro, Brazil, 2001.

[3] de O. Braga, C., Haeusler, E. H., Meseguer, J., Mosses, P. D.: Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic, *AMAST 2000*, LNCS 1816, Springer, 2000.

[4] de O. Braga, C., Haeusler, E. H., Meseguer, J., Mosses, P. D.: Mapping Modular SOS to Rewriting Logic, *LOPSTR 2002*, LNCS 2664, Springer, 2003.

[5] Doh, K.-G., Mosses, P. D.: Composing Programming Languages by Combining Action-Semantics Modules, *Science of Computer Programming*, **47**(1), 2003, 3–36.

[6] Hennessy, M.: *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*, Wiley, New York, 1990.

[7] Kahn, G.: Natural Semantics, *STACS'87*, LNCS 247, Springer, 1987.

[8] Liang, S., Hudak, P.: Modular Denotational Semantics for Compiler Construction, *ESOP'96*, LNCS 1058, Springer, 1996.

[9] Meseguer, J., de O. Braga, C.: Modular Rewriting Semantics of Programming Languages, *AMAST'04*, LNCS, Springer, 2004, To appear.

[10] Milner, R.: *Communication and Concurrency*, Prentice-Hall, 1989.

[11] Milner, R.: Operational and Algebraic Semantics of Concurrent Processes, in: *Handbook of Theoretical Computer Science* (J. van Leeuwen, Ed.), vol. B, chapter 19, Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[12] Milner, R., Tofte, M., Harper, R., MacQueen, D.: *The Definition of Standard ML (Revised)*, The MIT Press, 1997.

[13] Moggi, E.: *An Abstract View of Programming Languages*, Technical Report ECS-LFCS-90-113, Computer Science Dept., Univ. of Edinburgh, 1990.

[14] Mosses, P. D.: *Action Semantics*, Cambridge Tracts in Theoretical Computer Science 26, Cambridge University Press, 1992.

[15] Mosses, P. D.: *Foundations of Modular SOS*, BRICS RS-99-54, Dept. of Computer Science, Univ. of Aarhus, 1999, Full version of [16].

[16] Mosses, P. D.: Foundations of Modular SOS (Extended Abstract), *MFCS'99*, LNCS 1672, Springer, 1999.

[17] Mosses, P. D.: *A Modular SOS for Action Notation*, BRICS RS-99-56, Dept. of Computer Science, Univ. of Aarhus, 1999, Full version of [18].

[18] Mosses, P. D.: A Modular SOS for Action Notation (Extended Abstract), *AS'99*, BRICS NS-99-3, Dept. of Computer Science, Univ. of Aarhus, 1999.

[19] Mosses, P. D.: *A Modular SOS for ML Concurrency Primitives*, BRICS RS-99-57, Dept. of Computer Science, Univ. of Aarhus, 1999.

[20] Mosses, P. D.: Pragmatics of Modular SOS, *AMAST'02*, LNCS 2422, Springer, 2002.

[21] Mosses, P. D.: Modular Structural Operational Semantics, *JLAP*, 2004, To appear, special issue on SOS.

[22] Mosses, P. D., Watt, D. A.: The Use of Action Semantics, *Formal Description of Programming Concepts III*, North-Holland, 1987.

[23] Nielson, H. R., Nielson, F.: *Semantics with Applications: A Formal Introduction*, Wiley, Chichester, UK, 1992.

[24] Plotkin, G. D.: *A Structural Approach to Operational Semantics*, DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981, To appear in JLAP, special issue on SOS, 2004.

[25] Slonneger, K., Kurtz, B. L.: *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, Addison-Wesley, 1995.

[26] Watt, D. A.: *Programming Language Syntax and Semantics*, Prentice-Hall, 1991.

[27] Winskel, G.: *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993.