BRICS
Basic Research in Computer Science

BRICS RS-99-57 P. D. Mosses: A Modular SOS for ML Concurrency Primitives

A Modular SOS for ML Concurrency Primitives

Peter D. Mosses

BRICS Report Series RS-99-57
ISSN 0909-0878 December 1999

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/99/57/`

A Modular SOS for ML Concurrency Primitives *

Peter D. Mosses

BRICS and Department of Computer Science, University of Aarhus
Ny Munkegade, bldg. 540, DK-8000 Aarhus C, Denmark
Home page: `http://www.brics.dk/~pdm/`

Abstract. Modularity is an important pragmatic aspect of semantic descriptions. In denotational semantics, the issue of modularity has received much attention, and appropriate abstractions have been introduced, so that definitions of semantic functions may be independent of the details of how computations are modelled. In structural operational semantics (SOS), however, this issue has largely been neglected, and SOS descriptions of programming languages typically exhibit rather poor modularity—for instance, extending the described language may entail a complete reformulation of the description of the original constructs.
A proposal has recently been made for a modular approach to SOS, called MSOS. The basic definitions of the Modular SOS framework are recalled here, but the reader is referred to other papers for a full introduction.
This paper focusses on illustrating the applicability of Modular SOS, by using it to give a description of a sublanguage of Concurrent ML (CML); the transition rules for the purely functional constructs do not have to be reformulated at all when adding references and/or processes. The paper concludes by comparing the MSOS description with previous operational descriptions of similar languages.
The reader is assumed to be familiar with conventional SOS, with the concepts of functional programming languages such as Standard ML, and with fundamental notions of concurrency.

1 Conventional SOS

In the conventional SOS framework [23, 24] programs (and all their constituent phrases) are generally modelled as *labelled transition systems*:

Definition 1. *A* labelled transition system (LTS) *is a structure* $(\Gamma, T, \mathbb{A}, \longrightarrow)$, *where* Γ *is the set of* configurations, $T \subseteq \Gamma$ *is the set of* terminal configurations, $\mathbb{A}$ *is the set of* labels, *and* $\longrightarrow\ \subseteq \Gamma \times \mathbb{A} \times \Gamma$ *is the* transition relation. *For configurations* $\gamma, \gamma' \in \Gamma$ *and label* $\alpha \in \mathbb{A}$*, the assertion that* $(\gamma, \alpha, \gamma')$ *is in the transition relation is written* $\gamma \xrightarrow{\alpha} \gamma'$ *(implying* $\gamma \notin T$*).*

A computation *(from* γ*) is a sequence of transitions* $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ *, which is either infinite or finishes with a configuration* $\gamma' \in T$.

* Work done while visiting Computer Science Laboratory, SRI International, USA.

The main characteristic feature of SOS is that transition relations are specified inductively, according to the abstract syntax grammar, by giving sets of inference rules where the syntactic components of the initial configurations in the premises of a rule are generally components of that in the conclusion. Other formulae, such as equations, may be used as side-conditions on rules (although for notational convenience one may list them with the premises). The intended transition relation is the least such relation that is closed under the given inference rules.[1]

There are two distinct styles of SOS. In so-called *small-step SOS*, each transition in a computation generally corresponds to an indivisible bit of information processing, such as adding two computed numbers, or assigning a computed number to a variable. In *big-step SOS*, also known as *Natural Semantics* [10], a (terminating) computation is a single transition leading directly to a terminal configuration, corresponding to the composition of the computations of its constituent phrases. The big-step style can be formally regarded as a special case of the small-step style. The two styles may also be mixed in the same description. e.g., big-step for expression evaluation and small-step for command execution; alternatively, the transitive closure of the small-step transition relation can be used to represent the big-step relation [23]. (The small-step style is generally regarded as preferable for describing concurrent processes involving non-termination and interleaving, although a big-step treatment is also possible [22].)

Intermediate configurations in small-step SOS generally involve an extension of abstract syntax, allowing any sub-tree to be replaced by its computed value. Let us refer to the extended syntax as *value-added*. In some languages, the computed values can be identified with canonical terms of the original syntax, and then no extension is needed.

Configurations often involve familiar semantic components, such as stores that map variables to their assigned values. Environments (mapping identifiers to their denoted values) are however usually treated as separate arguments of a *relative* transition relation $\rho \vdash \gamma \xrightarrow{\alpha} \gamma'$ [10, 23]. Input, output, and synchronization signals are all generally recorded in the labels on transitions.

For detailed explanations of the conventional SOS framework, the reader is referred to [1, 8, 10, 21, 23–25, 28]. The lack of modularity in conventional SOS may be observed in many papers in the literature (e.g., [2]), see also the discussion in Sect. 5.

2 Modular SOS

Modular SOS (MSOS) [15, 14] is a particularly simple and uniform style of SOS. The essential idea is to use the *labels* on transitions to represent general *information processing* steps; the configurations merely keep track of the flow of control and data, and are restricted to value-added syntax.

[1] A more complicated definition is needed when negations of assertions of transitions are allowed in premises [6].

In a transition $\gamma \xrightarrow{\alpha} \gamma'$, the label α must itself determine the *state* of the processed information both before and after the step. Two such transitions are composable only when the state after the first and the state before the second are identical. This intuition is conveniently represented by regarding the labels as the arrows of a category, with the states as the objects of the category, and by requiring adjacent labels in computations to be composable in the category.

Definition 2. *A* category *consists of a set of* arrows $\alpha \in \mathbb{A}$*, a set of* objects $o \in |\mathbb{A}|$*, together with total operations* $\mathit{pre}, \mathit{post} : \mathbb{A} \to |\mathbb{A}|$, $\mathit{id} : |\mathbb{A}| \to \mathbb{A}$*, and a partial composition operation* $\cdot\,;\cdot : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$*, such that:*

- $\alpha_1 ; \alpha_2$ *is defined iff* $\mathit{post}(\alpha_1) = \mathit{pre}(\alpha_2)$*, and then* $\mathit{pre}(\alpha_1 ; \alpha_2) = \mathit{pre}(\alpha_1)$ *and* $\mathit{post}(\alpha_1 ; \alpha_2) = \mathit{post}(\alpha_2)$*;*
- $\cdot\,;\cdot$ *is associative, that is* $\alpha_1 ; (\alpha_2 ; \alpha_3) = (\alpha_1 ; \alpha_2) ; \alpha_3$ *when defined;*
- $\mathit{id}(\mathit{pre}(\alpha)) ; \alpha = \alpha = \alpha ; \mathit{id}(\mathit{post}(\alpha))$*;*
- $\mathit{pre}(\mathit{id}(o)) = o = \mathit{post}(\mathit{id}(o))$*.*

The objects $o = \mathit{pre}(\alpha)$ *and* $o' = \mathit{post}(\alpha)$ *are called the* source *and* target *of the arrow* α*, and may be indicated by writing* $\alpha : o \to o'$*; the arrow* $\mathit{id}(o)$ *is called the* identity *arrow for the object* o*. The subset of identity arrows of* $\mathbb{A}$ *is written* $\mathbb{I}^{\mathbb{A}}$*, or just* $\mathbb{I}$ *when* $\mathbb{A}$ *is evident; we let the variables* ι, ι', ι_1*, etc., range only over* $\mathbb{I}$*.*

The foundations for MSOS are provided by *arrow-labelled transition systems* (which, surprisingly, appears to be a novel combination of the familiar notions of LTS and category):

Definition 3. *An* arrow-labelled transition system (ALTS) *is a labelled transition system* $(\Gamma, T, \mathbb{A}, \longrightarrow)$*, where* $\mathbb{A}$ *is a* category*. The objects* $o \in |\mathbb{A}|$ *are called the* states *of the ALTS.*

A computation *in the ALTS (from* γ*) is a sequence of transitions* $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ *, which is either infinite or finishes with a configuration* $\gamma' \in T$*, and moreover such that all adjacent labels* α_i*,* α_{i+1} *in it are composable in the category* $\mathbb{A}$ *(i.e., the labels in a computation trace a path through* $\mathbb{A}$*).*

For a more detailed explanation of arrow-labelled transition systems, including their reduction to conventional LTS and definition of bisimulation equivalence, the reader is referred to [15].

3 Label Transformers

The label categories used in MSOS can be constructed by applications of three fundamental *label transformers*, starting from a trivial category. Essentially, labels obtained this way are tuples whose components can be set and accessed independently; each label transformer lifts the setting and accessing operations, as well as composition, to deal with the fresh component. The fundamental label transformers, defined in App. A, are analogous to some of the simpler monad transformers which are used in the monadic approach to denotational semantics.

The fundamental label transformers enjoy two important properties [15]:

– Different orders of application of the label transformers yield the same category (up to isomorphism).
– Applying a label transformer to a label category in an MSOS preserves computations (provided that the side conditions on the transition rules are insensitive to the transformation of labels).

Let INDEX be the set of indices i that may be used to refer to components of labels; typical elements of INDEX are *store* and *acts*, and differently-spelled indices are assumed to be distinct elements. Let UNIV be the universe whose elements u represent the information that may be processed; typical subsets of UNIV are STORE^2 and ACT^*. The partial functions $get(\alpha, i)$ and $set(\alpha, i, u)$ are everywhere undefined in the trivial category ***TrivCat***, and the various label transformers extend their domains of definition, as specified in App. A.

ContextInfo(i, E) adds an information component with values in E, indexed by i. The *value* of this component is *preserved* by label composition. Typically, E is a set of environments, mapping identifiers to denoted values.

MutableInfo(i, S) adds an information component with values in S, indexed by i. *Changes* to this component are *sequenced* by label composition. Typically, S is a set of stores, mapping addresses to assigned values. The following abbreviations (defined in App. A) are useful in connection with ***MutableInfo***(i, S): $get_{pre}(\alpha, i)$ accesses the state of the mutable information indexed i before a transition labelled α, and a transition labelled $set_{post}(\alpha, i, s')$ determines that its state is s' after the transition.

Finally, ***EmittedInfo***(i, A, f, τ) adds an information component with values in A, indexed by i. *Values* of this component are *composed by* f, and τ determines its value in identity labels $\iota \in \mathbb{I}$. Typically, (A, f, τ) is a free monoid, i.e., a set of sequences with τ being the empty sequence.

The definitions of the trivial label category and the three fundamental label transformers given in App. A are *completely independent* of the programming language whose MSOS is to be described. The transformers are used in an MSOS simply by mentioning their names and supplying the appropriate arguments. The index arguments of label transformers used in the same MSOS are assumed to be distinct. However, the same label transformer may be used more than once (with different index arguments).

4 Application of MSOS to a Subset of CML

Reppy [26, 27] has defined the operational semantics of λ_{cv}, a small concurrent λ-calculus that models the main concurrency features of CML. He uses the (non-structural) style of operational semantics based on reduction of *evaluation contexts*, as developed by Felleisen et al. [4, 33].

Here, we describe the same language λ_{cv} using MSOS, thus illustrating the applicability of our framework. To facilitate comparison with Reppy's semantics, we adhere closely to his choice of notation (despite some mild idiosyncrasies).[2]

[2] A previous version of this paper adopted the notation of [2], and described a slightly different language.

We also follow him by using some of the abstract syntax trees as computed values, by employing syntactic substitution rather than explicit environments (although we also illustrate the use of environments, see Appendix B), and by ignoring the types of expressions when giving the dynamic semantics of λ_{cv}.

The illustration of MSOS given below is in the small-step style. It is straightforward to reformulate the MSOS for expression evaluation in the big-step style (although then one should also define a predicate corresponding to divergence of expression evaluation). However, it seems best to keep to the small-step style for process evaluation, since the big-step treatment of concurrency appears to be relatively awkward [22].

Reppy's static semantics for λ_{cv} is given as a conventional big-step SOS (natural semantics), and could easily be reformulated in MSOS, using the ***ContextInfo*** label transformer to introduce type environments.

We first describe the purely functional part of λ_{cv}. Before extending it with concurrent processes and channels, we show how to add ML-style references, thus demonstrating the independence of the label transformers. (Reppy considers instead the representation of references using processes and channels, following Berry, Milner, and Turner [2].)

4.1 The Functional Fragment

The functional fragment of λ_{cv} described below is essentially Plotkin's original call-by-value language λ_v.

Abstract Syntax

$x \in \textsc{Var}$	variables
$c \in \textsc{Const} = \textsc{BConst} \cup \textsc{FConst}$	constants
$b \in \textsc{BConst} = \{\texttt{()}, \texttt{true}, \texttt{false}, \texttt{0}, \texttt{1}, \ldots\}$	base constants
$f \in \textsc{FConst} = \{\texttt{+}, \texttt{-}, \texttt{fst}, \texttt{snd}, \ldots\}$	function constants
$\textsc{Var} \cap \textsc{Const} = \emptyset$	
$e \in \textsc{Exp}$	expressions
$v \in \textsc{Val}$	values

$e ::= v$	value
$\mid x$	variable
$\mid e_1\ e_2$	application
$\mid \texttt{(}e_1\,\texttt{.}\,e_2\texttt{)}$	pair
$\mid \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2$	let
$v ::= c$	constant
$\mid \texttt{(}v_1\,\texttt{.}\,v_2\texttt{)}$	pair value
$\mid \lambda x\texttt{(}e\texttt{)}$	λ-abstraction

A pair of values $\texttt{(}v_1\,\texttt{.}\,v_2\texttt{)}$ may always be regarded as a value, despite the fact that it is also an expression according to the grammar.

Configurations

$$\begin{array}{ll} \gamma ::= e & \text{arbitrary} \\ \tau ::= v & \text{terminal} \end{array}$$

The set of all configurations $\gamma \in \Gamma$ thus consists of expressions e (including values), and that of terminal configurations $\tau \in T$ consists simply of the values v that can be computed by expressions.

Label Transformers No label transformers are needed here: the labels α are completely arbitrary. For instance, the label category $\mathbb{A}$ may be simply ***TrivCat***.

In the absence of explicit environments, we need a meta-level notation for the substitution of a value v for a variable x in an expression e. Let us adopt the notation $e[x \mapsto v]$, as defined by Reppy [27] (avoiding capture of free variables by use of Barendregt's convention).

Transition Rules

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1\ e_2 \xrightarrow{\alpha} e_1'\ e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{v_1\ e_2 \xrightarrow{\alpha} v_1\ e_2'} \tag{1}$$

$$\lambda x\texttt{(}e\texttt{)}\ v \xrightarrow{\iota} e[x \mapsto v] \tag{2}$$

$$\texttt{+ (0.1)} \xrightarrow{\iota} \texttt{1} \qquad \texttt{+ (1.1)} \xrightarrow{\iota} \texttt{2} \qquad \ldots \tag{3}$$

$$\texttt{fst (}v_1\texttt{.}v_2\texttt{)} \xrightarrow{\iota} v_1 \qquad \texttt{snd (}v_1\texttt{.}v_2\texttt{)} \xrightarrow{\iota} v_2 \tag{4}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{\texttt{(}e_1\texttt{.}e_2\texttt{)} \xrightarrow{\alpha} \texttt{(}e_1'\texttt{.}e_2\texttt{)}} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{\texttt{(}v_1\texttt{.}e_2\texttt{)} \xrightarrow{\alpha} \texttt{(}v_1\texttt{.}e_2'\texttt{)}} \tag{5}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2 \xrightarrow{\alpha} \texttt{let } x\texttt{=}e_1' \texttt{ in } e_2} \qquad \texttt{let } x\texttt{=}v \texttt{ in } e \xrightarrow{\iota} e[x \mapsto v] \tag{6}$$

Notice that the transition rules given above insist on left-to-right evaluation, as well as call by value. Static scopes for variable bindings are ensured by the meta-level substitution notation. Complete programs are assumed to have no free variables, and the transition rules ensure that all intermediate configurations that can arise also have no free variables, hence there is no need for a rule giving the value of a variable x that occurs as an expression.

4.2 An Imperative Extension

The following extension of Sect. 4.1 provides ML-style references.

Abstract Syntax

$$f \in \textsc{FConst} = \{\ldots, \texttt{ref}, \texttt{assign}, \texttt{deref}\} \quad \text{function constants}$$

Configurations

$$l \in \textsc{Loc} \quad \text{locations}$$

$$\begin{array}{lll}
\gamma & ::= e & \text{arbitrary} \\
\tau & ::= v & \text{terminal} \\
v & ::= \ldots & \\
 & \mid\ l & \text{location}
\end{array}$$

Label Transformers

$$\boldsymbol{\mathit{MutableInfo}}(\mathit{store}, \textsc{Store})$$

where:

$$\mathit{store} \in \textsc{Index}$$
$$s \in \textsc{Store} = \textsc{Loc} \xrightarrow{\mathit{fin}} \textsc{Val}$$

For stores, the notation $s[l \mapsto v]$ denotes the store that maps l to v, and otherwise maps locations l' to their values $s(l')$ according to s. (Stores cannot be expressions in the described language, so there should be no danger of confusing the notation $s[l \mapsto v]$ with Reppy's notation for substitution $e[x \mapsto v]$. Note however that when $l \notin \mathit{dom}(s)$, the store $s[l \mapsto v]$ is always different from s, whereas when x does not occur in e, the expression resulting from $e[x \mapsto v]$ is the same as e.)

Transition Rules

$$\frac{s = \mathit{get}_{\mathit{pre}}(\iota, \mathit{store}) \qquad l \notin \mathit{dom}(s) \qquad \alpha = \mathit{set}_{\mathit{post}}(\iota, \mathit{store}, s[l \mapsto v])}{\texttt{ref}\ v \xrightarrow{\alpha} l} \tag{7}$$

$$\frac{s = \mathit{get}_{\mathit{pre}}(\iota, \mathit{store}) \qquad l \in \mathit{dom}(s) \qquad \alpha = \mathit{set}_{\mathit{post}}(\iota, \mathit{store}, s[l \mapsto v])}{\texttt{assign}\ (l\,.\,v) \xrightarrow{\alpha} \texttt{()}} \tag{8}$$

$$\frac{s = \mathit{get}_{\mathit{pre}}(\iota, \mathit{store}) \qquad v = s(l)}{\texttt{deref}\ l \xrightarrow{\iota} v} \tag{9}$$

An application `ref` v not only returns a fresh location l, it also assigns v to it. Thus the domain of the store gives the set of locations that are in use.

The application `deref` l merely returns the value stored in l, the use of the identity label ι in (9) above ensuring that there can be no side-effects. (To describe the implicit dereferencing found in most imperative programming languages, one would give the rule for l instead of `deref` l.)

4.3 Concurrent Processes

The following extension of Sect. 4.1 (or of Sect. 4.2) completes the MSOS of λ_{cv}.

Abstract Syntax

$f \in$ FCONST $= \{\ldots,$ `choose`, `guard`, `never`, `receive`, `transmit`, `wrap`, `wrapAbort`$\}$	function constants
$p \in$ PROCS	processes
$e ::= \ldots$	
$\mid$ `chan` x `in` e	channel creation
$\mid$ `spawn` e	process creation
$\mid$ `sync` e	synchronization
$p ::= e$	single process

Configurations

$\gamma ::= e \mid p$	arbitrary
$\tau ::= v$	terminal
$k \in$ CHAN	channel names
$ev \in$ EVENT	event values
$v ::= \ldots$	
$\mid k$	channel name
$\mid ev$	event value
$\mid$ (`G` e)	guarded event
$ev ::= \Lambda$	never
$\mid k!v$	channel output
$\mid k?$	channel input
$\mid (ev \Rightarrow v)$	wrapper
$\mid (ev_1 \oplus ev_2)$	choice
$\mid (ev \mid v)$	abort wrapper

The above syntax and values are exactly as given by Reppy [26, 27]. Note that all the values added here are intermediate values, and they are not allowed to occur in the initial program.

$p ::= \ldots$	
$\mid p_1 \parallel p_2$	concurrent process

Our auxiliary syntax for concurrent processes above is similar to that of conventional process algebra (ACP, CCS, etc.), differing somewhat from Reppy's.

It would be straightforward to add a syntactic congruence on concurrent processes, essentially turning them into multi-sets:

$$p \parallel (p' \parallel p'') = (p \parallel p') \parallel p'' \qquad p \parallel p' = p' \parallel p.$$

Here, however, the congruence would only save us a single symmetric rule. A λ_{cv} program starts out as a single expression, and concurrent processes are created only by evaluation of `spawn` e.

Label Transformers

$$\textit{\textbf{MutableInfo}}(\mathit{chans}, \textsc{Chans})$$

where:

$$\begin{aligned} \mathit{chans} &\in \textsc{Index} \\ K &\in \textsc{Chans} = \mathbb{P}_{\mathit{fin}}(\textsc{Chan}) \quad \text{channel sets} \end{aligned}$$

$$\textit{\textbf{EmittedInfo}}(\mathit{acts}, \textsc{Act}^*, \mathit{concat}, [\,])$$

where:

$$\begin{array}{rll} \mathit{acts} &\in \textsc{Index} & \\ A &\in \textsc{Act}^* & \text{action sequences} \\ a &\in \textsc{Act} = \textsc{Sync} \cup \textsc{Spawn} & \text{actions} \\ (ev, e) &\in \textsc{Sync} = \textsc{Event} \times \textsc{Exp} & \text{synchronization possibilities} \\ v &\in \textsc{Spawn} = \textsc{Val} & \text{spawned processes} \end{array}$$

$\textsc{Act}^*$ is the set of finite sequences $a_1 \ldots a_n$ of elements $a_i \in \textsc{Act}$, with concatenation *concat* and the empty sequence $[\,]$ forming a monoid.

Transition Rules

Expressions

$$\texttt{never ()} \xrightarrow{\iota} \Lambda \tag{10}$$

$$\texttt{transmit (}k\texttt{.}v\texttt{)} \xrightarrow{\iota} k!v \tag{11}$$

$$\texttt{receive } k \xrightarrow{\iota} k? \tag{12}$$

$$\texttt{wrap (}ev\texttt{.}v\texttt{)} \xrightarrow{\iota} (ev \Rightarrow v) \tag{13}$$

$$\texttt{choose (}ev_1\texttt{.}ev_2\texttt{)} \xrightarrow{\iota} (ev_1 \oplus ev_2) \tag{14}$$

$$\texttt{wrapAbort (}ev\texttt{.}v\texttt{)} \xrightarrow{\iota} (ev \mid v) \tag{15}$$

The above applications all compute *events* $ev \in \textsc{Event}$, whereas those that follow compute *guarded events* of the form (`G` e). (If one removes the function `guard` from λ_{cv}, it becomes possible to regard all the remaining functional constants for events as *constructors*, cf. [2].)

$$\texttt{guard } v \xrightarrow{\iota} (\texttt{G (}v\texttt{ ()))} \tag{16}$$

$$\texttt{wrap ((G }e\texttt{).}v\texttt{)} \xrightarrow{\iota} (\texttt{G (wrap (}e\texttt{.}v\texttt{)))} \tag{17}$$

$$\texttt{choose ((G }e_1\texttt{).}ev_2\texttt{)} \xrightarrow{\iota} (\texttt{G (choose (}e_1\texttt{.}ev_2\texttt{)))} \tag{18}$$

$$\texttt{choose (}ev_1\texttt{.(G }e_2\texttt{))} \xrightarrow{\iota} (\texttt{G (choose (}ev_1\texttt{.}e_2\texttt{)))} \tag{19}$$

$$\texttt{choose ((G }e_1\texttt{).(G }e_2\texttt{))} \xrightarrow{\iota} (\texttt{G (choose (}e_1\texttt{.}e_2\texttt{)))} \tag{20}$$

$$\texttt{wrapAbort ((G }e\texttt{).}v\texttt{)} \xrightarrow{\iota} (\texttt{G (wrapAbort (}e\texttt{.}v\texttt{)))} \tag{21}$$

The evaluation of e in a guarded event (`G` e) is delayed until the latter occurs as the argument of `sync`. Once the argument of `sync` has been evaluated to an (unguarded) event ev, it gives rise to a label α that indicates the *possibility* of a synchronization:

$$\texttt{sync (G}\ e\texttt{)} \xrightarrow{\iota} \texttt{sync}\ e \qquad \frac{\alpha = set(\iota, acts, (ev, e))}{\texttt{sync}\ ev \xrightarrow{\alpha} e} \tag{22}$$

The nondeterministic choice of e in the right-hand rule in (22) above is resolved by the matching of event values in (26) below. This nondeterminism could be eliminated by using a place-holder (or fresh variable) instead of e above, then changing (26) below so that the expressions determined by event matching are substituted for the place-holders in the two processes.

$$\frac{\begin{array}{c} K = get_{pre}(\iota, chans) \quad k \notin K \\ \alpha = set_{post}(\iota, chans, K \cup \{k\}) \end{array}}{\texttt{chan}\ x\ \texttt{in}\ e \xrightarrow{\alpha} \texttt{let}\ x\texttt{=}k\ \texttt{in}\ e} \tag{23}$$

The fresh channel k is recorded as having been allocated by adding it to the set K of channels in use.

One could use $e[x \mapsto k]$ instead of the `let`-expression in (23) above. The (slight) advantage of the formulation chosen here is that it is independent of whether bindings have been described using substitution or environments (cf. Appendix B).

$$\frac{\alpha = set(\iota, acts, v)}{\texttt{spawn}\ v \xrightarrow{\alpha} \texttt{()}} \tag{24}$$

`spawn` v merely signals that a new process is to be started to evaluate the application v `()`, cf. (27) below. (In a well-typed λ_{cv} program, v here is always a λ-abstraction of type `unit -> unit`.)

Processes

$$\frac{p_1 \xrightarrow{\alpha} p_1'}{p_1 \parallel p_2 \xrightarrow{\alpha} p_1' \parallel p_2} \qquad \frac{p_2 \xrightarrow{\alpha} p_2'}{p_1 \parallel p_2 \xrightarrow{\alpha} p_1 \parallel p_2'} \tag{25}$$

The above rules allow the evaluation of each process to proceed independently, in any order. (An expression e is a special case of a process p, and the labels for expression transitions are here taken to be the same as for process transitions, so there is no need to give a rule explicitly extending sequential evaluation to concurrent evaluation.)

$$\frac{\begin{array}{c} p_1 \xrightarrow{\alpha_1} p_1' \quad p_2 \xrightarrow{\alpha_2} p_2' \\ \alpha_1 = set(\iota, acts, (ev_1, e_1)) \quad \alpha_2 = set(\iota, acts, (ev_2, e_2)) \\ ev_1 \overset{k}{\asymp} ev_2\ \mathit{with}\ (e_1, e_2) \end{array}}{p_1 \parallel p_2 \xrightarrow{\iota} p_1' \parallel p_2'} \tag{26}$$

The only possibility for a single transition to involve more than one process is when two processes are both evaluating applications of `sync` to (unguarded) events, say ev_1 and ev_2, and the two events match. The predicate:

$$ev_1 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_1, e_2)$$

holds when the events ev_1 and ev_2 match (on channel k) with respective results e_1 and e_2. For instance:

- $k\,\texttt{!}\,v \overset{k}{\asymp} k\texttt{?}\ \mathit{with}\ (\texttt{()}, v)$ holds; but
- $k\,\texttt{!}\,v \overset{k}{\asymp} k\,\texttt{!}\,v\ \mathit{with}\ (e_1, e_2))$ does not hold for any e_1, e_2, and
- neither does $k\,\texttt{!}\,v \overset{k}{\asymp} k'\texttt{?}\ \mathit{with}\ (e_1, e_2)$ when $k \neq k'$.

Reppy's definition of event matching [26, 27] is reproduced in Appendix C.

The labels α_1 and α_2 in (26) above are always identity ι apart from the *acts* component, since a synchronization in the language considered here cannot arise together with observable changes to the mutable information.

$$\frac{e \xrightarrow{\alpha} e' \qquad \alpha = \mathit{set}(\iota, \mathit{acts}, v)}{e \xrightarrow{\iota} e' \parallel (v\ \texttt{()})} \tag{27}$$

The above rule deals with the spawning of processes. It is only applicable when e is an entire process, since a term of the form $e' \parallel e''$ cannot occur as an expression. (Generalizing e, e' to p, p' in (27) above would allow spawning to be handled at any level of the process structure, but without any observable differences.) As for synchronization in (26), the above rule relies on the fact that process spawning cannot occur together with observable changes to the mutable information.

$$p \parallel v \xrightarrow{\iota} p \tag{28}$$

The above rule discards the values of spawned processes. The evaluation of a complete program reaches a terminal configuration only after all spawned processes have terminated and been discarded.

The only transitions allowed for complete programs are those whose label α satisfies $\mathit{get}(\alpha, \mathit{acts}) = [\,]$.

That concludes the illustrative MSOS of λ_{cv}. Notice especially that with MSOS, the extensions of the functional fragment with references and with concurrent processes are completely independent, and the order of making the extensions is irrelevant.

Following Reppy [26, 27], we should now eliminate *unfair* computations from the specified transition system for complete programs, thus requiring implementations of λ_{cv} to let ready processes make progress, and not ignore possible synchronizations on any particular channel indefinitely. It would also be useful to check that our MSOS could be extended to a description of the full CML language. In a different direction, one might follow Ferreira, Hennessy, and Jeffrey [5, 9] by trying to establish a theory of equivalence for λ_{cv} expressions. These and other topics are left for future work, since our main purpose in the present paper is merely to illustrate the applicability of MSOS to the description of ML concurrency constructs.

5 Related Work

Here, we focus on comparing our MSOS for λ_{cv} with other published descriptions of similar languages. A more general assessment of the relation of MSOS to other work is provided elsewhere [15]; see also [16]

5.1 Using Evaluation Contexts

It is rather straightforward to compare the transition rules of our MSOS for the functional fragment of λ_{cv} with the evaluation context reduction semantics given by Reppy [26, 27]: those transition rules that merely propagate transitions to enclosing constructs, e.g., (1) above, correspond to alternatives of his grammar for evaluation contexts, and the axioms that make unobservable reductions, e.g., (4) above, correspond to his reduction rules.

Reppy does not consider the extension of the functional fragment with references, but proposes the representation of references by processes, following [2]. Thus that part of his description is not directly comparable to our MSOS.

Instead of using a conventional binary combinator for concurrent processes such as $e_1 \parallel e_2$, Reppy considers sets of processes tagged with identifiers, written $\langle \pi_1; e_1 \rangle + \ldots + \langle \pi_n; e_n \rangle$. The main difference, however, is that in MSOS we follow process algebra descriptions by letting the labels carry synchronization and spawning possibilities *upwards* through the syntactic structure, whereas with evaluation contexts one looks *downwards* through the structure for the occurrences of particular values.

On the basis of these two descriptions of λ_{cv}, it seems that the evaluation-context framework may have the edge over MSOS concerning conciseness: the grammar for evaluation contexts is a much more compact description of the flow of control than our rules for propagating transitions to enclosing constructs. On the other hand, these same MSOS rules can be adapted straightforwardly to describe the flow of processed information through constructs, which generally requires separate reduction rules when using evaluation contexts. On the whole, the modularity of the two descriptions of λ_{cv} appears to be equally good.

One significant advantage of MSOS over evaluation contexts concerns the *compositionality* of the transition rules, as pointed out by Ferreira, Hennessy, and Jeffrey [5, 9]. With MSOS, the transitions for a construct are determined by the transitions for its components, together with the form of the construct itself. This should allow us to provide a bisimulation and prove some useful equivalences for λ_{cv}, following the work of Ferreira, Hennessy, and Jeffrey.

5.2 Using Conventional SOS

A conventional SOS for a language $\mu\mathrm{CML}^{cv}$, which corresponds closely to λ_{cv}, has recently been published by Ferreira, Hennessy, and Jeffrey [5], see also Jeffrey [9]. The authors provide a theory of weak bisimulation for their language; they also extract an LTS from Reppy's semantics, and prove that this LTS is bisimilar to the LTS defined by their SOS.

There are some points of similarity between the SOS for $\mu\mathrm{CML}^{cv}$ and our MSOS for λ_{cv}. In particular, Ferreira et al. also use a binary parallel composition operator ($e_1 \mathrel{+\!\!\!\!\!\mapsto} e_2$) to record the configurations of spawned processes, and exploit labels α to propagate synchronization possibilities upwards through the process structure. However, their treatment of values and spawned processes is such that in a transition $e \xrightarrow{\surd v} e'$, the computed value v is in the *label* and e' then gives the spawned process (or just the unit value `()` when there is none). This is essentially the *opposite* of the treatment in the present paper where, in $e \xrightarrow{\alpha} e'$, the label α holds any process to be spawned, and e' may be a computed value. The main motivation given for their somewhat surprising approach is that "a CML process has a *main thread of control*, and only the main thread can return a value" [5, p. 453]. However, Reppy himself abstracts from this rather incidental detail of CML in his own semantics for λ_{cv}, and even considers letting terminated processes "evaporate" [27, p. 84], so the cited motivation seems unconvincing.

Unfortunately, the treatment of computed values and spawned processes that Ferreira et al. have adopted leads to rather awkward rules for transitions, e.g. for pairs:

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{(e_1, e_2) \xrightarrow{\alpha} (e_1', e_2)} \qquad \frac{e_1 \xrightarrow{\surd v} e_1'}{(e_1, e_2) \xrightarrow{\alpha} e_1' \mathrel{+\!\!\!\!\!\mapsto} \texttt{let } x\texttt{=}v \texttt{ in } \langle x, e_2\rangle} \tag{29}$$

and for let-expressions:

$$\frac{e_1 \xrightarrow{\tau} e_1'}{\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2 \xrightarrow{\tau} \texttt{let } x\texttt{=}e_1' \texttt{ in } e_2} \qquad \frac{e_1 \xrightarrow{\surd v} e_1'}{\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2 \xrightarrow{\tau} e_1' \mathrel{+\!\!\!\!\!\mapsto} e_2[v/x]} \tag{30}$$

The SOS rules given for the functional constructs of $\mu\mathrm{CML}^{cv}$ (as illustrated above) are not what one would expect in the absence of the concurrency constructs, so the modularity of this conventional SOS is clearly quite poor; moreover, when comparing $\mu\mathrm{CML}^{cv}$ with λ_{cv}, Ferreira et al. conjecture that the description "would need to be considerably altered" to cope with `guard` and `wrapAbort`, which are absent in $\mu\mathrm{CML}^{cv}$. Jeffrey [9] also admits that "there are some problems with this semantics"—but he uses this as a motivation to consider a variant of $\mu\mathrm{CML}^{cv}$ (with computation types), rather than to re-engineer the description of the existing language.

The development of a comparable theory of bisimulation for λ_{cv} based on MSOS is left to future work.

5.3 Mixing Evaluation Contexts and SOS

Berry, Milner, and Turner [2] have given a description of a language similar to the languages considered by Reppy [26, 27] and by Ferreira et al. [5]. For the functional fragment, they provide a simple conventional SOS, corresponding closely to that given in the present paper (but omitting labels). All the rules for the functional fragment undergo major (albeit systematic) reformulation when concurrency constructs are added, demonstrating poor modularity—as previously noted by the present author and Musicante [19]. For example, the rules for pairing become:

$$\frac{K, P[p : e_1] \xrightarrow{S} K', P'[p : e_1']}{K, P[p : (e_1, e_2)] \xrightarrow{S} K', P'[p : (e_1', e_2)]} \tag{31}$$

$$\frac{K, P[p : e_2] \xrightarrow{S} K', P'[p : e_2']}{K, P[p : (v_1, e_2)] \xrightarrow{S} K', P'[p : (v_1, e_2')]} \tag{32}$$

where $[p : e]$ denotes a singleton process set, $P[p : e]$ denotes $P \cup [p : e]$ (*not* substitution into a context), K is the set of allocated channels, and S is the set of processes involved in the transition that it labels. Notice that syntactic components of the configuration in the premises of the rules are *not* components of those of the conclusions, i.e., the transition relation is not defined inductively according to the abstract syntax. The description of concurrent constructs is quite comparable to Reppy's, even though there are no explicit evaluation contexts here.

Berry et al. also show how to extend the functional fragment with references: by changing configurations from e to e, s, and reformulating *all* the previously-given rules (thereby giving a particularly clear demonstration of the poor modularity of conventional SOS). The main contribution of their paper consists of proofs that their extension of the functional fragment with concurrency constructs preserves the semantics of sequential expressions (corresponding to a general result about label transformers in MSOS [15]) and in showing that a particular representation of stores by processes corresponds to the direct introduction of references.

The stated aim "to incorporate the semantics of this language with the semantics of SML" [2, p. 119] has apparently not yet been achieved—perhaps because of modularity problems, or because the definition of SML [11] uses the big-step style of SOS? Interestingly, the recent alternative definition of SML proposed by Harper and Stone [7] is based on a translation from SML into an "internal language" whose operational semantics is given as a reduction system for evaluation contexts; the integration of concurrency constructs in that definition seems more likely to be feasible. It appears that ML2000 [29] is to include (an asynchronous version of) CML events.

5.4 Using Enhanced Operational Semantics

The technique of incorporating all semantic information in the labels has been proposed as a general principle for SOS by Degano and Priami [3], and exploited to obtain parametricity in their *Enhanced Operational Semantics* (EOS). Priami [25, Ch. 8] uses EOS to describe Facile (a language very similar to CML). Many of the transition rules given for the Facile constructs are identical to our MSOS rules for CML constructs, modulo minor notational details. Moreover, the EOS description extends smoothly, without reformulation, from function expressions to behaviour expressions, and finally to distributed behaviour expressions.

The main difference between EOS and MSOS lies in the set of labels: in EOS, it is essentially the set of *proof terms* $\theta \in \Theta$ for transitions. Such proof terms are generated from elementary actions by operations that record the structure of the process that executes them. Auxiliary functions are defined on proof terms to extract various kinds of information, e.g., $\ell(\theta)$ returns the action of θ. (In the Facile example, also a "quite technical" auxiliary relation is used in connection with spawning processes on particular nodes, but this would not be needed in connection with an EOS for CML.)

Another difference concerns computations: EOS is based on the ordinary LTS framework, where the labels on adjacent transitions do not restrict computations. In MSOS, however, labels that represent changes to a store can only be adjacent in a computation when the store resulting from the first transition is the same as that before the second transition. Perhaps an EOS dealing with an updatable store would have to give an ad hoc definition of computations? EOS has good modularity for describing concurrency at different levels of abstraction, but it is difficult to assess its overall modularity in the absence of examples of the treatment of side-effects.

It is debatable which of EOS and MSOS is the more general: EOS provides all possible information in the labels, and the relevant items are extracted using auxiliary functions; MSOS provides just the information that has been included using label transformers, together with fixed auxiliary operations to set and get each item separately. It is any case straightforward to provide support for EOS in MSOS: all that is needed is to include the label transformer ***EmittedInfo***(*proof*, Θ), and to set this component of the label appropriately in the conclusions of the relevant transition rules.

5.5 Using Action Semantics

The present author and Musicante [19] have given an action semantics [12, 13, 20, 32] for the same language as described by Berry et al. [2]. An action semantics translates the described programming language into an action notation, which has a fixed semantics, defined using (a notational variant of) small-step SOS [12, Appendix C]—essentially the same technique as exploited by Harper and Stone in their alternative definition of SML [7]. The design of action notation is based on the notion of orthogonal facets of information processing (facets are related to monad transformers, as shown by Wansbrough and Hamer [30, 31]);

the use of the primitives and combinators of action notation in the description of a construct is independent of which facets are needed for the description of other constructs. In particular, the description of the functional fragment does not require any reformulation, merely extension, when concurrent processes are added. Thus the modularity of action semantics is just as good as that of MSOS.

One problem with action semantics has been that the original SOS of action notation was not only expressed in an unconventional notational variant of SOS, but also its modularity was rather poor. This has recently been remedied by using MSOS to define the semantics of action notation [18].

So, which is better: to describe the operational semantics of a programming language directly, using MSOS, or indirectly, using action semantics?

The main advantage of the action semantics approach is that the combinators of action notation provide concise abbreviations for particular *patterns* of transition rules. For instance, the combinator for sequential action performance (written A_1 `then` A_2 in standard action notation) abbreviates the pattern of transitions that occurs in the MSOS rules in Section 4.1 for left-to-right evaluation of applications (1) and pairs (4). A further advantage would show up in connection with the description of ML-style exceptions: action notation provides a primitive for escaping from normal action performance (with a value), and a combinator for trapping such escapes; in (small-step) MSOS, the propagation of the exception value through all the syntactic constructs apart from the exception handler has to be specified explicitly.

However, MSOS also has some advantages over action semantics. Perhaps the main one is that the only new notation provided by the MSOS framework is that for the label transformers, whereas the full standard action notation is quite rich, and becoming familiar with it requires a significant initial investment of effort. Another drawback of action semantics stems from the very generality of action notation: its equational theory is too weak to be of much practical use. With MSOS, one may be able to prove stronger properties, exploiting awareness of the exact patterns of transitions and configurations that can arise in the semantics of a particular programming language.

6 Conclusion

This paper has demonstrated that MSOS is applicable to languages such as CML. It remains to be seen whether MSOS provides an appropriate basis for proving properties of CML programs, and for establishing a satisfactory theory of bisimulation (or testing) equivalence. In any case, the evident high degree of modularity of MSOS descriptions allows their easy extension, modification, and partial re-use, and thereby greatly facilitates the practical application of SOS for documenting language design decisions—solving a problem left open by Plotkin [23].

Acknowledgements Thanks to Christiano Braga, Pierpaolo Degano, Alexander Knapp, Søren B. Lassen, Ugo Montanari, Mark-Oliver Stehr, and Carolyn Talcott for useful comments on earlier versions of this paper. Thanks also to José

Meseguer for helpful suggestions concerning the presentation of label transformers in App. A. The author is supported by BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark; by an International Fellowship from SRI International; and by DARPA-ITO through NASA-Ames contract NAS2-98073.

References

1. E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 51–136. Springer-Verlag, 1991.
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. P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, June 1996.
4. M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 193–217. North-Holland, 1987.
5. W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisumulation for core CML. *J. Functional Programming*, 8(5):447–451, 1998.
6. W. Fokkink and R. van Glabbeek. Ntyft/ntyxt rules reduce to ntree rules. *Information and Computation*, 126(1):1–10, 1996.
7. R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Robin Milner Festschrifft*. MIT Press, 1998. To appear.
8. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, New York, 1990.
9. A. S. A. Jeffrey. Semantics for core concurrent ML using computation types. In A. D. Gordon and A. J. Pitts, editors, *Higher Order Operational Techniques in Semantics, Proceedings*. Cambridge University Press, 1997.
10. 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.
11. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
12. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
13. P. D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, 1996.
14. P. D. Mosses. Foundations of modular SOS. Research Series BRICS-RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/54`. Full version of [15].
15. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available [14].

16. P. D. Mosses. Logical specification of operational semantics. In *CSL'99, Proc. Conf. on Computer Science Logic*, volume 1683 of *LNCS*, pages 32–49. Springer-Verlag, 1999. Also available at `http://www.brics.dk/RS/99/55`.
17. P. D. Mosses. A modular SOS for Action Notation. Research Series BRICS-RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/56`. Full version of [18].
18. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In P. D. Mosses and D. A. Watt, editors, *AS'99, Proc. Second International Workshop on Action Semantics, Amsterdam, The Netherlands*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, May 1999. Full version available [17].
19. P. D. Mosses and M. A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *LNCS*, pages 461–479. Springer-Verlag, 1994.
20. P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.
21. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, Chichester, UK, 1992.
22. A. M. Pitts and J. R. X. Ross. Process calculus based upon evaluation to committed form. *Theoretical Comput. Sci.*, 195:155–182, 1998.
23. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
24. G. D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*. IFIP, North-Holland, 1983.
25. C. Priami. *Enhanced Operational Semantics for Concurrency*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Pisa, 1996.
26. J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.
27. J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Computer Science Dept., Cornell Univ., 1992. Tech. Rep. TR 92-1285.
28. K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.
29. The ML 2000 Working Group. Principles and a preliminary design for ML2000. Draft, `http://www.luca.demon.co.uk/Bibliography.html`, Mar. 1999.
30. K. Wansbrough. A modular monadic action semantics. Master's thesis, Dept. of Computer Science, Univ. of Auckland, Feb. 1997.
31. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997.
32. D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.
33. A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Dept. of Computer Science, Rice University, 1991.

A Fundamental Label Transformers

The label categories constructed by label transformers are *product categories*, and the various components of the labels can be inspected and changed independently of each other. To avoid dependence on the order in which transformers are composed, particular components are referred to via symbolic indices $i \in \textsc{Index}$. All components of labels are taken from some universe $\textsc{Univ}$.

In the trivial label category, the labels have no components at all:

Definition 4. *The category* $\boldsymbol{TrivCat}$ *has a single object and a single arrow, and the operations*

$$\begin{aligned} get &: \boldsymbol{TrivCat} \times \textsc{Index} \to \textsc{Univ} \\ set &: \boldsymbol{TrivCat} \times \textsc{Index} \times \textsc{Univ} \to \boldsymbol{TrivCat} \end{aligned}$$

are completely undefined.

Definition 5. *Let* $\mathbb{B}$ *be a category, and* $i \in \textsc{Index}$*. Then the label transformer* $\boldsymbol{LabTrans}(i, \mathbb{B})$ *maps any label category* $\mathbb{A}$ *to* $\mathbb{A} \times \mathbb{B}$*, and extends the operations*

$$\begin{aligned} get &: \mathbb{A} \times \textsc{Index} \to \textsc{Univ} \\ set &: \mathbb{A} \times \textsc{Index} \times \textsc{Univ} \to \mathbb{A} \end{aligned}$$

from $\mathbb{A}$ *to* $\mathbb{A} \times \mathbb{B}$ *by defining*

$$\begin{aligned} get((\alpha, u), j) &= \begin{cases} u, & \text{if } i = j \\ get(\alpha, j), & \text{otherwise} \end{cases} \\ set((\alpha, u), j, u') &= \begin{cases} (\alpha, u'), & \text{if } i = j \\ (set(\alpha, j, u'), u), & \text{otherwise.} \end{cases} \end{aligned}$$

For any $\mathbb{A}$, $\mathbb{B}$ the projection from $\mathbb{A} \times \mathbb{B}$ to $\mathbb{A}$ is a functor. Moreover, for any object $b \in |\mathbb{B}|$ the embedding that maps objects $a \in |\mathbb{A}|$ to (a, b) and arrows $\alpha \in \mathbb{A}$ to $(\alpha, id(b))$ is also a functor.

The following label categories and their associated label transformers are of fundamental significance:

Definition 6. *For any set* E *let* $\boldsymbol{Discrete}(E)$ *be the discrete category having the elements of* E *as objects (the only arrows being the identity arrows). Let* $\boldsymbol{ContextInfo}(i, E)$ *be* $\boldsymbol{LabTrans}(i, \boldsymbol{Discrete}(E))$*.*

Typically, E above is a set of environments, and the use of $\boldsymbol{ContextInfo}(i, E)$ makes the current environment available in labels at index i.

Definition 7. *For any set* S *let* $\boldsymbol{Pairs}(S)$ *be the category having the elements of* S *as objects, where for* $s, s' \in S$ *the only arrow with source* s *and target* s' *is represented by the pair* (s, s')*. Let* $\boldsymbol{MutableInfo}(i, S)$ *be* $\boldsymbol{LabTrans}(i, \boldsymbol{Pairs}(S))$*.*

Typically, S above is a set of stores, and the use of $\boldsymbol{MutableInfo}(i, S)$ makes *pairs* of stores available in labels. For inspecting the source store and setting the target store of a label, the following auxiliary operations are convenient: when $get(\alpha, i) = (s, s')$, let $get_{pre}(\alpha, i) = s$ and $set_{post}(\alpha, i, s'') = set(\alpha, i, (s, s''))$.

Definition 8. *For any monoid A with associative operation $f : A \times A \to A$ and unit τ, let $\boldsymbol{Monoid}(A, f, \tau)$ be the category with just one object, and with the elements of A as its only arrows, composition being determined by f and the identity arrow by τ. Let $\boldsymbol{EmittedInfo}(i, A, f, \tau)$ be $\boldsymbol{LabTrans}(i, \boldsymbol{Monoid}(A, f, \tau))$.*

Typically, A above is the free monoid of sequences generated by some set of signals, with f being sequence concatenation and τ being the empty sequence; then the use of $\boldsymbol{EmittedInfo}(i, A, f, \tau)$ makes sequences of signals available in labels.

B Modular SOS Using Environments

B.1 A Functional Fragment

Abstract Syntax

$x \in \textsc{Var}$	variables
$c \in \textsc{Const} = \textsc{BConst} \cup \textsc{FConst}$	constants
$b \in \textsc{BConst} = \{\texttt{()}, \texttt{true}, \texttt{false}, \texttt{0}, \texttt{1}, \ldots\}$	base constants
$f \in \textsc{FConst} = \{\texttt{+}, \texttt{-}, \texttt{fst}, \texttt{snd}, \ldots\}$	function constants
$\textsc{Var} \cap \textsc{Const} = \emptyset$	
$e \in \textsc{Exp}$	expressions
$v \in \textsc{Val}$	values

$e ::= v$	value
$\mid\ x$	variable
$\mid\ e_1\ e_2$	application
$\mid\ (e_1 \,.\, e_2)$	pair
$\mid\ \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2$	let
$\mid\ \lambda x(e)$	λ-abstraction
$v ::= c$	constant
$\mid\ (v_1 \,.\, v_2)$	pair value

Configurations

$\gamma ::= e$	arbitrary
$\tau ::= v$	terminal
$e ::= \ldots$	
$\mid\ e\ \texttt{with}\ \rho$	expression closure
$v ::= \ldots$	
$\mid\ (x\texttt{,}e\texttt{,}\rho)$	function closure

Notice that compared to Section 4.1, a λ-abstraction is no longer a value, and both function and expression closures have been introduced—the former to hold the static bindings ρ of λ-abstractions, the latter to keep hold of the bindings while evaluating the bodies of λ-abstractions.[3]

[3] Plotkin [23] was able to avoid introducing expression closures because there was already a general form of local declaration in his example language; they are not needed in big-step SOS either [10].

Label Transformers

$$\textbf{\textit{ContextInfo}}(\mathit{env}, \textsc{Env})$$

where:

$$\mathit{env} \in \textsc{Index}$$
$$\rho \in \textsc{Env} = \textsc{Var} \overset{\mathit{fin}}{\to} \textsc{Val}$$

The notation $\rho[x \mapsto v]$ below denotes the environment that maps x to v, and otherwise gives the same results as ρ.

Transition Rules

$$\frac{\rho = \mathit{get}(\iota, \mathit{env}) \qquad v = \rho(x)}{x \xrightarrow{\iota} v} \tag{33}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1\ e_2 \xrightarrow{\alpha} e_1'\ e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{v_1\ e_2 \xrightarrow{\alpha} v_1\ e_2'} \tag{34}$$

$$\frac{\rho = \mathit{get}(\iota, \mathit{env})}{\lambda x\texttt{(}e\texttt{)} \xrightarrow{\iota} \texttt{(}x\texttt{,}e\texttt{,}\rho\texttt{)}} \tag{35}$$

$$\texttt{(}x\texttt{,}e\texttt{,}\rho\texttt{)}\ v \xrightarrow{\iota} e \ \texttt{with}\ \rho[x \mapsto v] \tag{36}$$

$$\frac{\alpha' = \mathit{set}(\alpha, \mathit{env}, \rho') \qquad e \xrightarrow{\alpha'} e'}{e \ \texttt{with}\ \rho' \xrightarrow{\alpha} e' \ \texttt{with}\ \rho'} \tag{37}$$

$$v \ \texttt{with}\ \rho' \xrightarrow{\iota} v \tag{38}$$

$$\texttt{+ (0.1)} \xrightarrow{\iota} \texttt{1} \qquad \texttt{+ (1.1)} \xrightarrow{\iota} \texttt{2} \qquad \ldots \tag{39}$$

$$\texttt{fst (}v_1\texttt{.}v_2\texttt{)} \xrightarrow{\iota} v_1 \qquad \texttt{snd (}v_1\texttt{.}v_2\texttt{)} \xrightarrow{\iota} v_2 \tag{40}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{\texttt{(}e_1\texttt{.}e_2\texttt{)} \xrightarrow{\alpha} \texttt{(}e_1'\texttt{.}e_2\texttt{)}} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{\texttt{(}v_1\texttt{.}e_2\texttt{)} \xrightarrow{\alpha} \texttt{(}v_1\texttt{.}e_2'\texttt{)}} \tag{41}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{\texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 \xrightarrow{\alpha} \texttt{let}\ x\texttt{=}e_1'\ \texttt{in}\ e_2} \tag{42}$$

$$\frac{\rho = \mathit{get}(\alpha, \mathit{env}) \qquad \alpha' = \mathit{set}(\alpha, \mathit{env}, \rho[x \mapsto v]) \qquad e_2 \xrightarrow{\alpha'} e_2'}{\texttt{let}\ x\texttt{=}v\ \texttt{in}\ e_2 \xrightarrow{\alpha} \texttt{let}\ x\texttt{=}v\ \texttt{in}\ e_2'} \tag{43}$$

$$\texttt{let}\ x\texttt{=}v\ \texttt{in}\ v' \xrightarrow{\iota} v' \tag{44}$$

Notice that rules (34), (39)–(42) are as before. Moreover, no changes at all are needed to Sections 4.2 and 4.3.

C Event Matching

The matching of event values is written '$ev_1 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_1, e_2)$', and defined to be the smallest relation satisfying the following inference rules, exactly as in [26, 27]:

$$k!v \overset{k}{\asymp} k? \ \mathit{with}\ (\texttt{()}, v) \tag{45}$$

$$\frac{ev_1 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_1, e_2)}{ev_2 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_2, e_1)} \tag{46}$$

$$\frac{ev_1 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_1, e_2)}{ev_2 \overset{k}{\asymp} (ev_2 \Rightarrow v) \ \mathit{with}\ (e_1, v\ e_2)} \tag{47}$$

$$\frac{ev_1 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_1, e_2) \qquad e_3 = \mathrm{AbortAct}(ev_3)}{ev_2 \overset{k}{\asymp} (ev_2 \oplus ev_3) \ \mathit{with}\ (e_1, \texttt{(}e_3\texttt{;}\ e_2\texttt{)})} \tag{48}$$

$$\frac{ev_1 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_1, e_2) \qquad e_3 = \mathrm{AbortAct}(ev_3)}{ev_2 \overset{k}{\asymp} (ev_3 \oplus ev_2) \ \mathit{with}\ (e_1, \texttt{(}e_3\texttt{;}\ e_2\texttt{)})} \tag{49}$$

$$\frac{ev_1 \overset{k}{\asymp} ev_2 \ \mathit{with}\ (e_1, e_2)}{ev_2 \overset{k}{\asymp} (ev_2 \mid v) \ \mathit{with}\ (e_1, e_2)} \tag{50}$$

The syntax $\texttt{(}e_1\texttt{;}\ e_2\texttt{)}$ represents sequencing, formally abbreviating the application $\texttt{snd (}e_1\texttt{.}e_2\texttt{)}$. The function AbortAct : $\textsc{Event} \to \textsc{Exp}$ used above is defined inductively as follows:

$$\begin{aligned}
\mathrm{AbortAct}(\Lambda) &= \texttt{()} \\
\mathrm{AbortAct}(k?) &= \texttt{()} \\
\mathrm{AbortAct}(k!v) &= \texttt{()} \\
\mathrm{AbortAct}(ev \Rightarrow e) &= \mathrm{AbortAct}(ev) \\
\mathrm{AbortAct}(ev_1 \oplus ev_2) &= \texttt{(}\mathrm{AbortAct}(ev_1)\texttt{;}\ \mathrm{AbortAct}(ev_2)\texttt{)} \\
\mathrm{AbortAct}(ev \mid v) &= \texttt{(}\mathrm{AbortAct}(ev)\texttt{;}\ \texttt{spawn}\ v\texttt{)}
\end{aligned}$$

Recent BRICS Report Series Publications

RS-99-57 **Peter D. Mosses.** ***A Modular SOS for ML Concurrency Primitives*****. December 1999. 22 pp.**

RS-99-56 **Peter D. Mosses.** ***A Modular SOS for Action Notation*****. December 1999. 39 pp. Full version of paper appearing in Mosses and Watt, editors,** ***Second International Workshop on Action Semantics*****, AS '99 Proceedings, BRICS Notes Series NS-99-3, 1999, pages 131–142.**

RS-99-55 **Peter D. Mosses.** ***Logical Specification of Operational Semantics*****. December 1999. 18 pp. Invited paper. Appears in Flum, Rodríguez-Artalejo and Mario, editors,** ***European Association for Computer Science Logic: 13th International Workshop*****, CSL '99 Proceedings, LNCS 1683, 1999, pages 32–49.**

RS-99-54 **Peter D. Mosses.** ***Foundations of Modular SOS*****. December 1999. 17 pp. Full version of paper appearing in Kutyłowski, Pacholski and Wierzbicki, editors,** ***Mathematical Foundations of Computer Science: 24th International Symposium*****, MFCS '99 Proceedings, LNCS 1672, 1999, pages 70–80.**

RS-99-53 **Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen.** ***Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using UPPAAL*****. December 1999. 9 pp.**

RS-99-52 **Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar, and P. S. Thiagarajan.** ***Towards a Theory of Regular MSC Languages*****. December 1999.**

RS-99-51 **Olivier Danvy.** ***Formalizing Implementation Strategies for First-Class Continuations*****. December 1999. Extended version of an article to appear in** ***Programming Languages and Systems: Ninth European Symposium on Programming*****, ESOP '00 Proceedings, LNCS, 2000.**

RS-99-50 **Gerth Stølting Brodal and Srinivasan Venkatesh.** ***Improved Bounds for Dictionary Look-up with One Error*****. December 1999. 5 pp.**