BRICS

Basic Research in Computer Science

Modular Structural Operational Semantics

Peter D. Mosses

BRICS Report Series RS-05-7

ISSN 0909-0878 February 2005

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/05/7/`

Modular Structural Operational Semantics

Peter D. Mosses

BRICS & Department of Computer Science, University of Aarhus
Aabogade 34, DK-8200 Aarhus N, Denmark[1]

Abstract

Modular SOS (MSOS) is a variant of conventional Structural Operational Semantics (SOS). Using MSOS, the transition rules for each construct of a programming language can be given incrementally, once and for all, and do not need reformulation when further constructs are added to the language. MSOS thus provides an exceptionally high degree of modularity in language descriptions, removing a shortcoming of the original SOS framework.

After sketching the background and reviewing the main features of SOS, the paper explains the crucial differences between SOS and MSOS, and illustrates how MSOS descriptions are written. It also discusses standard notions of semantic equivalence based on MSOS. An appendix shows how the illustrative MSOS rules given in the paper would be formulated in conventional SOS.

Key words: structural operational semantics, SOS, modularity, MSOS

1 Introduction

Modular Structural Operational Semantics (MSOS) [23] is a variant of the conventional Structural Operational Semantics (SOS) framework [30]. Using MSOS, the transition rules for each construct of a programming language can be given incrementally, once and for all, and generally do not need reformulation when further constructs are added to the described language.

MSOS solves the modularity problem for SOS as effectively as monad transformers do for denotational semantics. Moreover, although the foundations

Email address: `pdmosses@brics.dk` (Peter D. Mosses).
URL: `www.brics.dk/~pdm` (Peter D. Mosses).

[1] BRICS: Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

of MSOS involve concepts from Category Theory, MSOS descriptions can be understood just as easily as ordinary SOS, and MSOS has been class-tested successfully at Aarhus in undergraduate courses.

Previous papers have presented the foundations of MSOS [22,23], discussed its pragmatic aspects [28], and demonstrated its usefulness in modular operational descriptions of action notation [25] and the core of Concurrent ML [26]. The present paper gives a comprehensive presentation of MSOS, incorporating some notational improvements. To facilitate comparison between MSOS and SOS, the illustrative language constructs described here in MSOS are all from Plotkin's notes on SOS [30], and an appendix shows how they would be formulated in conventional SOS.

1.1 Background

SOS, introduced by Plotkin in his seminal Aarhus lecture notes in 1981 [30], is a well-known framework that can be used for specifying the semantics of concurrent systems [1,16] and programming languages [17]. It has been widely taught, especially at the undergraduate level [12,29,37,41], and it is generally found to be significantly more accessible than denotational semantics.

However, conventional SOS descriptions of programming languages have quite poor modularity. This is apparent already in the examples given by Plotkin in his notes [30]: e.g., the initial description of simple arithmetic expressions needs to be reformulated three times when adding variable identifiers and value identifiers to expressions (first separately and then both together). The required reformulations are actually quite routine, but that doesn't lessen their undesirability.

Plotkin himself admitted that "As regards modularity we just hope that if we get the other things in a reasonable shape, then current ideas for imposing modularity on specifications will prove useful" (remarks at end of Chapter 2, op. cit.). More than a decade later, however, "the other things" in SOS appeared to be in a very reasonable shape – but there was no sign of any improvement at all regarding modularity. When extending a pure functional language with concurrency primitives and/or references, for instance, the SOS rules for all the functional constructs had to be completely reformulated [5].

In denotational semantics, language descriptions originally suffered from similar problems regarding poor modularity. These problems were largely solved by Moggi's introduction of monads and monad transformers [18] (although Plotkin and Power have recently proposed to generate monads by algebraic operations instead of constructing them by monad transformers [31–33]). Action semantics [19,20] is an alternative approach to obtaining good modularity in denotational descriptions, and the action notation used for expressing action denotations includes combinators that correspond closely to monadic composition. However, the reference definition of action notation [19] was originally formulated in SOS, and has poor modularity; to define subsets and extensions of action notation would have required extensive reformulation of the SOS rules.

In 1997, Wansbrough and Hamer [39,40] suggested to replace the SOS of action notation by a monadic semantics, primarily to improve its modularity. The author was thus faced with a dilemma: either to abandon SOS and adopt the proposed modular monadic semantics of action notation, or to try to improve the modularity of SOS. Following the latter course, the basic ideas for MSOS emerged while studying Plotkin's notes and trying to avoid the various reformulations that are to be found there.

1.2 Overview

Section 2 reviews the main features of SOS at some length, drawing attention to various technical details concerning Plotkin's notes and recalling notation. Section 3 explains the fundamental differences between SOS and MSOS, and illustrates how examples of SOS from Plotkin's notes are formulated in MSOS. Section 4 briefly discusses notions of semantic equivalence for MSOS. Section 5 relates MSOS to some other frameworks. Section 6 concludes, mentioning some topics left for future work.

Readers are assumed to be familiar with the basic notions of operational semantics, and with the standard conceptual analysis of common constructs of high-level programming languages. Although MSOS involves the notion of a category, familiarity with Category Theory is not required. The notation used here generally follows Plotkin's notes regarding common features of SOS and MSOS, to facilitate comparison, although this gives rise to some stylistic differences from the notation used in previous (and probably also future) papers on MSOS.

2 Conventional SOS

In the original SOS framework, as introduced in Plotkin's notes, the operational semantics of a programming language is represented by a transition system. The configurations (or states) of the transition system always involve the syntax of programs and their parts (commands, declarations, expressions, etc.); they may also involve computed values, and abstract representations of other information, such as bindings and stored values. Transitions may be labelled. The transition relation between configurations is specified inductively, by simple and conditional rules. Usually, conditional rules are structural, in that when the conclusion of a rule is a transition for a compound syntactic construct, the conditions involve transitions only for its components. Computations, consisting of sequences of transitions, represent executions of programs.

Let's now review these main features of SOS in more detail, drawing attention to some relatively subtle technical points that are perhaps not so clear from Plotkin's notes, and which will be quite significant in connection with MSOS in Section 3.

2.1 Syntax

SOS descriptions of programming languages start from abstract syntax. Specifications of abstract syntax introduce symbols for syntactic sets, meta-variables ranging over those sets, and notation for constructor functions. Some of the syntactic sets are usually regarded as basic, and left open or described only informally. Meta-variables can be distinguished by primes and subscripts.

The notation for abstract syntax constructor functions is conventionally specified by context-free grammars, presented in a style reminiscent of BNF. The terminal symbols used in the grammars are usually chosen to be highly suggestive of the concrete syntax of the language being described, and the non-terminal symbols can simply be the meta-variables.

Table 1 specifies abstract syntax for various constructs taken from Plotkin's notes, following his style of notation rather closely. Such a specification determines a many-sorted algebraic signature, and the syntactic sets together with the constructors form a free algebra generated by the basic sets. The elements of the algebra can be regarded as trees. Sometimes binary constructors are specified to be commutative and/or associative; then the elements are essentially equivalence classes of trees.

Well-formedness constraints on programs, such as declaration before use and type-correctness, are usually ignored in abstract syntax. The full operational semantics of a program can be regarded as a composition of its static semantics (corresponding to compile-time checks of well-formedness) and its dynamic semantics (corresponding to run-time computation). Both static and dynamic semantics can be specified in SOS, based on the same abstract syntax; here, we shall focus on dynamic semantics.

Table 1
Abstract syntax of some illustrative constructs

$$
\begin{array}{llcl}
\textbf{Truth-values:} & t \in \mathbb{T} & = & \{\mathrm{tt}, \mathrm{ff}\} \\
\textbf{Numbers:} & n \in \mathbb{N} & = & \{0, 1, 2, \ldots\} \\
\textbf{Identifiers:} & x \in \mathrm{Id} & = & \{\mathrm{x}_0, \mathrm{x}_1, \mathrm{x}_2, \ldots\} \\
\textbf{Binary ops.:} & \mathit{bop} \in \mathrm{Bop} & = & \{+, -, *, \ldots\}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{Constants:} & \mathit{con} \in \mathrm{Con} \\
& \quad \mathit{con} ::= t \mid n \\
\textbf{Expressions:} & e \in \mathrm{Exp} \\
& \quad e ::= \mathit{con} \mid x \mid e_0 \ \mathit{bop} \ e_1 \mid \mathbf{let}\, d\, \mathbf{in}\, e \\
\textbf{Commands:} & c \in \mathrm{Com} \\
& \quad c ::= \mathbf{nil} \mid x := e \mid c_0 \,;\, c_1 \mid d \,;\, c \mid \\
& \qquad \mathbf{if}\, e\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \mid \mathbf{while}\, e\, \mathbf{do}\, c \\
\textbf{Declarations:} & d \in \mathrm{Dec} \\
& \quad d ::= \mathbf{const}\, x = e \mid \mathbf{var}\, x := e \mid d_0 \,;\, d_1
\end{array}
$$

2.2 Computed Values

The SOS of most constructs of programming languages involves computations which, on termination, result in a value of some kind. For expressions, the computed values might be truth-values or numbers. A command can be regarded as computing a fixed, null value. It is natural also to regard a declaration as computing an environment, representing the bindings made by the declaration. Computed values, like all other entities in SOS, are supposed to be finite.

It's sometimes convenient to use the same elements both in abstract syntax and as computed values. For instance, the elements of the basic "syntactic" sets of truth-values and numbers in Table 1 may be the usual abstract mathematical booleans and integers, not requiring any evaluation at all. In the other direction, the empty command **nil** can be used also as the null value computed by a command, and abstract syntax for types (not illustrated here) could be used as the type-values computed by expressions in static semantics. For declarations, however, the environments that they compute aren't syntactic by nature, so here the computed values are disjoint from the abstract syntax. Table 2 illustrates how sets of computed values are specified. (The set of environments Env is defined in the next section.)

The idea of distinguishing a set of computed values for each kind of syntactic construct is prevalent in the monadic approach to denotational semantics [18], and can be related to earlier work by Reynolds on a general approach to

Table 2
Sets of computed values

Expression values:	$\mathbb{N} \cup \mathbb{T}$
Command values:	$\{\mathbf{nil}\}$
Declaration values:	Env

types for programs and their phrases [36]. Plotkin's notes were not entirely systematic regarding sets of computed values: commands were not regarded as computing any values at all, for instance.

2.3 Auxiliary Entities

Various auxiliary entities are needed in SOS, for use as computed values or as other components of configurations. For our illustrative examples here, we'll need (natural) numbers, (boolean) truth-values, environments, and stores. The numbers and truth-values were already introduced as basic sets in Table 1, and we'll follow Plotkin in using conventional mathematical notation for the associated operations. Environments $\rho \in \mathrm{Env}$ and stores $\sigma \in \mathrm{Stores}$ are finite functions, where the set of finite functions from X to Y is written $X \rightarrow_{\mathrm{fin}} Y$. The range of environments is written DVal (for "denotable" values), and that of stores SVal (for "storable" values). The set Loc of locations, representing (independent) memory cells, is left open. [2]

Table 3
Sets of auxiliary entities

Environments:	$\rho \in \mathrm{Env} = \mathrm{Id} \rightarrow_{\mathrm{fin}} \mathrm{DVal}$
Denotable values:	$\mathrm{DVal} = \mathbb{N} \cup \mathbb{T} \cup \mathrm{Loc}$
Stores:	$\sigma \in \mathrm{Stores} = \mathrm{Loc} \rightarrow_{\mathrm{fin}} \mathrm{SVal}$
Locations:	$l \in \mathrm{Loc}$ (arbitrary)
Storable values:	$\mathrm{SVal} = \mathbb{N} \cup \mathbb{T}$

Sets of finite functions with particular domains of definition can be introduced, e.g., Env_V for finite $V \subseteq \mathrm{Id}$ and Stores_L for finite $L \subseteq \mathrm{Loc}$. Moreover, both identifiers and locations can be associated with types of values, and attention restricted to type-preserving finite functions. In general, keeping track of domains of definition and types requires a considerable amount of tedious indexing, as illustrated from Section 2.5 onwards in Plotkin's notes. Such indexing, however, is not essential in SOS, and here, we'll make do without it.

[2] For pedagogical reasons, Plotkin's notes initially don't distinguish between identifiers and locations, using the set Var for both.

Application of a finite function $f \in X \to_{\text{fin}} Y$ to an arbitrary argument $x \in X$ is written as usual, $f(x)$, but note that the result of the application may be undefined. Plotkin's notes don't formalize the treatment of undefinedness. Astesiano [2] provides a coherent approach to dealing with undefinedness in connection with SOS, but a more general framework for (first-order) logical specifications supporting the use of partial functions has subsequently been provided by the Common Algebraic Specification Language, CASL [3,7]. The following paragraph summarizes the relevant features of CASL.

(Meta-)variables in terms are always interpreted as (defined) values, and the logic is 2-valued: a formula either holds or it doesn't, even when some terms in it have undefined values. When the value of any argument is undefined, the result of a function application is undefined, and the application of a predicate never holds. Equations may be either existential (holding only when the values of both terms are defined) or strong (holding also when the values of both terms are undefined), but the two kinds of equations are equivalent if one term is simply a variable or a defined constant (which will always be the case in this paper). The assertion $\mathrm{def}(t)$ merely insists on the definedness of the term t. The value of the partial constant 'undef' (not provided by, but specifiable in CASL) is undefined. Finally, when ϕ is any formula and t_0, t_1 are terms, the term '$t_0 \,\mathrm{when}\, \phi \,\mathrm{else}\, t_1$' is equivalent to t_0 when ϕ holds, and to t_1 otherwise.

Adopting the above understanding of how undefinedness is treated, we can specify the following notation for use in expressing finite functions:

Singleton: $x \mapsto y$ is the element of $X \to_{\text{fin}} Y$ determined by

$$(x \mapsto y)(x') = y \text{ when } (x = x') \text{ else undef}$$

for all $x, x' \in X$ and $y \in Y$. [3]

Overriding: $f[g]$ is the element of $X \to_{\text{fin}} Y$ determined by

$$f[g](x) = g(x) \text{ when } \mathrm{def}(g(x)) \text{ else } f(x)$$

for all $f, g \in X \to_{\text{fin}} Y$ and $x \in X$.

Domain of definition: For any element f of $X \to_{\text{fin}} Y$, $\mathrm{dom}(f)$ is the set of values $x \in X$ for which the application $f(x)$ is defined.

For instance, for any $\sigma \in \mathrm{Store}, l \in \mathrm{Loc}$, and $v \in \mathrm{SVal}$, $\sigma[l \mapsto v]$ expresses the store which maps l to v, and maps all other locations l' to the result (when defined) of the application $\sigma(l')$.

[3] In Plotkin's notes, $x \mapsto y$ is written $\{x = y\}$, or often just $x = y$.

2.4 Configurations for SOS

Configurations are states of transition systems, and computations consist of sequences of transitions between configurations, starting from an initial configuration, and terminating (if at all) in a final configuration.

An initial configuration for a computation of a part of a program consists of the syntax of that part, generally accompanied by auxiliary components. A final configuration generally has the same structure as an initial configuration, but with a computed value in place of the original syntax. (As previously mentioned, commands can be treated as computing a fixed null value.)

In the usual style of SOS, computations proceed gradually by small steps through intermediate configurations where some parts of the syntactic component have been replaced by their computed values. When the computed values are already included in abstract syntax, as with the truth-values or numbers computed by expressions or the null value **nil** computed by commands, the intermediate configurations that may arise are automatically included in the set of possible initial configurations, as are the final configurations.

In other cases, such as for declarations, it's necessary to generalize the sets of configurations. Following Plotkin, we specify this generalization by extending the grammar for abstract syntax with productions involving meta-variables ranging over computed values. Let's refer to the result of adding computed values in this way to syntactic sets as *value-added syntax*. Not only are the computed values thereby included in the corresponding syntactic sets, but also these sets are closed up under the syntactic constructor functions. Essentially, the sets of added computed values are treated in just the same way as the basic syntactic sets.

A precise definition would involve details of signatures and freely-generated algebras. An example should suffice: Table 4 specifies value-added syntax for declarations, extending the abstract syntax specified in Table 1. The meta-variable ρ ranges over Env (see Table 3), and the effect of the production is to embed Env in Dec.

Table 4
Value-added syntax

Declarations: $d ::= \rho$

The separation of the production $d ::= \rho$ from the other productions for d makes a clear distinction between the original abstract syntax and the value-added syntax. Note however that the meta-variable d now ranges not only over the original declarations, but also over environments, and arbitrary mixtures of declarations and environments.

Once the required value-added syntax has been specified, the sets of configurations Γ and final configurations T can be defined. Γ always involves the abstract syntax of the programming language, and T involves the sets of computed values. In Plotkin's notes, set comprehension is used to define Γ and T, as illustrated in Table 5.

Table 5
Configurations for SOS

$$\Gamma = \{\langle \rho, e, \sigma \rangle\} \cup \{\langle \rho, c, \sigma \rangle\} \cup \{\langle \rho, d, \sigma \rangle\}$$
$$T = \{\langle \rho, con, \sigma \rangle\} \cup \{\langle \rho, \mathbf{nil}, \sigma \rangle\} \cup \{\langle \rho, \rho', \sigma \rangle\}$$

2.5 Transition Systems for SOS

In SOS, the operational semantics of a programming language is modelled by a *transition system* (together with some notion of *equivalence*, see Section 4). Plotkin defined several kinds of transition system, differing with regard to whether the set of final configurations is distinguished, and whether transitions are labelled. The most general kind is called a *labelled terminal transition system*:

Definition 1 *A* labelled terminal transition system *LTTS is a quadruple $\langle \Gamma, A, \longrightarrow, T \rangle$ consisting of a set Γ of configurations γ, a set A of labels α, 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 LTTS (from γ_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 γ_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 $\alpha_1 \ldots \alpha_n \gamma_n$.*

An SOS specification of a programming language consists of definitions of the sets Γ, A, and T, together with a set of rules specifying the transition relation $\longrightarrow$. Optionally, for each syntactic set S, the relevant subsets of Γ, A, and T can be identified.

In SOS, transition relations are specified inductively, by rules. A rule is formed from assertions of transitions $t \xrightarrow{t'} t''$, where the terms t, t', t'' can contain meta-variables.

A *simple rule* consists of a single assertion $t \xrightarrow{t'} t''$. It specifies that $\gamma \xrightarrow{\alpha'} \gamma''$ holds for all triples $\langle \gamma, \alpha', \gamma'' \rangle$ that result from evaluating the terms t, t', t'' with the same interpretation of their common meta-variables. Note that application of a partial function outside its domain of definition always leads to the value of the enclosing term being undefined, and a transition relation cannot hold on undefined arguments.

A *conditional rule* is written:

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

A simple rule can be regarded as a conditional rule with an empty list of conditions. The conditions $c_1, \ldots, c_n$ and the conclusion c are each assertions of transitions. The rule specifies that whenever all the conditions c_i hold for a particular interpretation of the meta-variables that occur in the rule, so does the conclusion c.

Given a set of rules, a triple $\langle \gamma, \alpha', \gamma'' \rangle$ is in the specified transition relation if and only if a finite upwardly-branching tree can be formed satisfying the following conditions:

(1) all nodes are labelled by elements of $\Gamma \times A \times \Gamma$,
(2) the root is labelled by $\langle \gamma, \alpha', \gamma'' \rangle$, and
(3) for each n-ary node in the tree, there is a rule $\frac{c_1, \ldots, c_n}{c}$ and an interpretation of the meta-variables that occur in it, such that the label of the node is the interpretation of c, and the labels of the branches are the interpretations of $c_1, \ldots, c_n$, taken in any order.

The syntactic parts of the configuration terms t in assertions $t \xrightarrow{t'} t''$ play a particularly significant rôle in SOS. Let's refer to them as the "controls" of the transitions. A rule is called *structural* when the controls of its conditions all specify components (i.e. subtrees) of the control of its conclusion. SOS doesn't require that rules are structural, but in practice, they often are.

Side-conditions can be added to both simple and conditional rules. They don't involve the transition relation: they are typically equations, set memberships, or definedness assertions. Side-conditions are often written together with the ordinary conditions, rather than displayed separately, since they can easily be distinguished from transition assertions. Negations of side-conditions can be used freely.

The rules given in Table 6 below illustrate an SOS specification of a transition relation for evaluation of arithmetic expressions in a purely functional language. Assuming that identifiers are bound directly to constant values, stores are redundant here, and hence omitted from the configurations. Conventionally, transitions $\langle\rho, e\rangle \longrightarrow \langle\rho, e'\rangle$, where the environment ρ always remains the same, are written $\rho \vdash e \longrightarrow e'$. (Plotkin suggested the use of *relative* transition systems, and generally omitted environments when defining sets of configurations.)

Table 6
SOS rules

$$\frac{\rho \vdash e_0 \longrightarrow e_0'}{\rho \vdash e_0 \; bop \; e_1 \longrightarrow e_0' \; bop \; e_1} \tag{2}$$

$$\frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash con_0 \; bop \; e_1 \longrightarrow con_0 \; bop \; e_1'} \tag{3}$$

$$\frac{bop = +, \quad n = n_0 + n_1}{\rho \vdash n_0 \; bop \; n_1 \longrightarrow n} \tag{4}$$

$$\frac{\rho(x) = con}{\rho \vdash x \longrightarrow con} \tag{5}$$

Rules (2) and (3) enforce sequential evaluation of e_0 and e_1. Interleaving of the evaluations can be allowed simply by using e_0 instead of con_0 in (3).

A rule similar to (4) is needed for each element of Bop. The variables *bop* and n are introduced to avoid the ambiguities that would arise if we were to specify $\rho \vdash n_0 + n_1 \longrightarrow n_0 + n_1$; such extra variables (and the side-conditions that define them) aren't needed when the elements of Bop are written differently from the corresponding mathematical operations on $\mathbb{N}$.

Notice that (5) gives rise to a transition only when ρ and x are such that $\rho(x) \in \text{Con}$, which can hold only when $x \in \text{dom}(\rho)$.

The transition relation specified by a set of rules is the least relation that satisfies all the rules. It always exists – regardless of whether the rules are structural or not. Structural induction can be used to prove properties of the specified transition relation when all the specified rules are structural. It is also possible to regard the rules themselves as a formal proof system, and then to reason about the structure of derivations of transitions.

The use of rules to specify relations in SOS isn't restricted to transition relations. In connection with static semantics, relations such as $\rho \vdash e : \tau$, asserting that e has type τ for a typing environment ρ, are specified in much the same way. In dynamic semantics, auxiliary predicates such as $c\surd$, asserting the possibility of termination of commands c, can be specified together with the transition relations. Thus the general case is that both conditions and conclusions of rules can be assertions involving any of the relations being specified.

Rules can also be generalized to allow negations of assertions as conditions – but then considerable care is needed to define what relation (if any) is specified by a set of rules [1]. Negations of side-conditions can be used freely.

2.7 Styles of SOS

SOS allows different styles of operational semantics. The style used in Table 6 above, where each step of a computation for an expression corresponds to an application of a single operation in some sub-expression, is called *small-step* SOS. At the other extreme is *big-step* SOS (also known as *natural semantics* [13]), which is illustrated in Table 7 below.

Table 7
Big-step SOS rules

$$\frac{\rho \vdash e_0 \longrightarrow n_0, \quad \rho \vdash e_1 \longrightarrow n_1}{\rho \vdash e_0 + e_1 \longrightarrow n_0 + n_1} \tag{6}$$

$$\rho \vdash con \longrightarrow con \tag{7}$$

$$\frac{\rho(x) = con}{\rho \vdash x \longrightarrow con} \tag{8}$$

An assertion of the form $\rho \vdash e \longrightarrow n$ holds when e can compute the value n in the environment ρ. It resembles the transition assertion $\rho \vdash e \longrightarrow e'$ (abbreviating $\langle\rho, e\rangle \longrightarrow \langle\rho, e'\rangle$ in the small-step style). However, a big-step SOS is not usually interpreted as a transition system: $\rho \vdash e \longrightarrow n$ is simply a ternary relation, and specified by rules just like any other relation. Evaluation goes straight from configurations involving abstract syntax to configurations involving computed values, so there is no need for value-added syntax in the big-step style. Notice that rules like (7), arising due to the use of computed values also in abstract syntax, are actually incompatible with the defining property of the set of final configurations in an LTTS.

Both the small- and big-step styles can be used together in the same SOS: big-step for expressions and small-step for commands, for example. Alternatively, the transitive closure of a small-step transition relation can be used to reduce a (terminating) multi-step computation to a single step (as illustrated throughout Plotkin's notes).

In general, the small-step style tends to require a greater number of rules than the big-step style, but this is outweighed by the fact that the small-step rules also tend to be simpler (each rule usually has at most one condition, except in connection with synchronization of transitions between concurrent processes). The small-step style facilitates the description of interleaving. Furthermore, it accurately reflects non-termination possibilities by infinite computations, whereas a big-step SOS simply ignores non-termination possibilities. Note also that big-step rules for loops and function applications are inherently non-structural, so it isn't possible to use structural induction for proving properties of the big-step SOS of languages that include such constructs.

On the other hand, when the semantics to be modelled is without side-effects and non-termination possibilities – being essentially just mathematical evaluation – the big-step style seems preferable; this is generally the case for static semantics, for instance, and for evaluation of literal constants in dynamic semantics.

We'll return to the issue of the small- and big-step styles at the end of Section 3. It turns out that interleaving can (somewhat surprisingly) be specified in the big-step style, whereas the small-step style has a definite advantage regarding the specification of the errors and exceptions.

2.8 Modularity in SOS

As mentioned in the Introduction, conventional SOS descriptions of programming languages have poor modularity, and adding further constructs to the described language may require a major reformulation of the rules for the previous constructs.

For instance, consider again the rules given in Table 6 above, specifying the evaluation of pure arithmetic expressions. Suppose that we are to extend the described language with commands, and to allow the inspection of stored values in expressions. Clearly, the store must now be included in the configurations for expression evaluation (as specified in Table 5) and the rules have to be changed accordingly. The revised rules are shown in Table 8 below, together with an extra rule (13) for inspecting stored values.

The specified rules require that expressions don't have side-effects. One might therefore abbreviate $\rho \vdash \langle e, \sigma \rangle \longrightarrow \langle e', \sigma \rangle$ to $\rho, \sigma \vdash e \longrightarrow e'$. However, when expressions are subsequently extended with constructs (such as function application) that allow side-effects, a reformulation to use assertions of the form

Table 8
SOS rules, reformulated

$$\frac{\rho \vdash \langle e_0, \sigma \rangle \longrightarrow \langle e'_0, \sigma \rangle}{\rho \vdash \langle e_0 \ bop \ e_1, \sigma \rangle \longrightarrow \langle e'_0 \ bop \ e_1, \sigma \rangle} \tag{9}$$

$$\frac{\rho \vdash \langle e_1, \sigma \rangle \longrightarrow \langle e'_1, \sigma \rangle}{\rho \vdash \langle con_0 \ bop \ e_1, \sigma \rangle \longrightarrow \langle con_0 \ bop \ e'_1, \sigma \rangle} \tag{10}$$

$$\frac{bop = +, \quad n = n_0 + n_1}{\rho \vdash \langle n_0 \ bop \ n_1, \sigma \rangle \longrightarrow \langle n, \sigma \rangle} \tag{11}$$

$$\frac{\rho(x) = con}{\rho \vdash \langle x, \sigma \rangle \longrightarrow \langle con, \sigma \rangle} \tag{12}$$

$$\frac{\rho(x) = l, \quad \sigma(l) = con}{\rho \vdash \langle x, \sigma \rangle \longrightarrow \langle con, \sigma \rangle} \tag{13}$$

$\rho \vdash \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ becomes unavoidable.

Similar evidence of the poor modularity of conventional SOS can be found throughout Plotkin's notes. Furthermore, extending expressions with concurrency constructs (to allow spawning of processes, and synchronization with expressions in other processes) would require the introduction of explicit labels on transitions, necessitating further reformulation of the specified rules, as illustrated by Berry et al. [5].

Apart from the need for occasional major reformulation of rules during the development of an SOS, there are further reasons for dissatisfaction with rules like those given in Table 8:

(1) The repetition of ρ and σ is *tedious*, and a clumsy way of specifying that environments are *generally* inherited by sub-expressions, whereas store updates follow the flow of control.
(2) The rules are *not reusable*, and formulated differently when describing the same construct occurring in different languages.

Regarding point (1), the *Definition of Standard ML* [17] introduces a "store convention" to avoid the repetitious mention of σ in big-step MSOS rules: the order in which the conditions of an abbreviated rule are written determines how σ's should be inserted to generate the real rule. However, if it's important to be able to avoid the repetitions of σ, this should be incorporated in the formalism, and not left to ad hoc conventions introduced in connection with particular language descriptions.

As for point (2), it would clearly be beneficial for authors to be able to *reuse* existing descriptions of common constructs when developing semantic descriptions. Provided that such reuse were made apparent (e.g., by explicit reference to uniquely-named modules) readers would also benefit, as they could see immediately that particular constructs have the same semantics in different languages. Moreover, with reusable rules it should be possible to prove properties (such as bisimulation equivalences) once-and-for-all for a set of common constructs, instead of re-proving them for each new language in which they occur.

The modular variant of SOS introduced in Section 3 eliminates both the above causes of dissatisfaction with SOS descriptions, at minimal notational cost, and without resorting to semi-formal conventions.

2.9 Abrupt Termination

One further issue affecting modularity in SOS concerns the description of constructs involving "abrupt termination" (errors, exceptions, breaks, goto's). Plotkin illustrated a straightforward way of dealing with dynamic errors: add an extra **error** configuration, and "error rules" that allow error configurations to propagate through each construct of the language. Plotkin's error propagation rules were all conditional rules; Table 9 illustrates how to get almost the same effect with only simple propagation rules, by treating **error** as a computed value. (This alternative technique gives rise to some extra transitions when computations lead to errors, but that needn't bother us here.)

Table 9
SOS for dynamic errors

Expression values: $\ldots \cup \{\mathbf{error}\}$

Value-added syntax: $e ::= \ldots \cup \mathbf{error}$

Final configurations: $T = \ldots \cup \{\langle \rho, \mathbf{error}, \sigma \rangle\}$

$$\frac{bop = -, \quad n = n_0 - n_1}{\rho \vdash \langle n_0 \; bop \; n_1, \sigma \rangle \longrightarrow \langle n, \sigma \rangle} \tag{14}$$

$$\frac{bop = -, \quad n_0 < n_1}{\rho \vdash \langle n_0 \; bop \; n_1, \sigma \rangle \longrightarrow \langle \mathbf{error}, \sigma \rangle} \tag{15}$$

$$\rho \vdash \langle \mathbf{error} \;\; bop \; e_1, \sigma \rangle \longrightarrow \langle \mathbf{error}, \sigma \rangle \tag{16}$$

$$\rho \vdash \langle con_0 \; bop \;\; \mathbf{error}, \sigma \rangle \longrightarrow \langle \mathbf{error}, \sigma \rangle \tag{17}$$

Similar, but more complicated, error propagation rules would be needed in a corresponding big-step SOS. (The *Definition of Standard ML* [17] managed to leave them implicit by introducing an "exception convention", based on the order in which the conditions of rules are written.)

Adding error propagation rules doesn't require reformulation of the original rules. However, the need to give extra rules for constructs which aren't themselves directly concerned with errors can be seen as further evidence of poor modularity in SOS. Moreover, the extra rules tend to roughly double the size of an SOS.

In Section 3.7, we'll illustrate a novel technique that eliminates the need for error propagation rules altogether. The technique, however, is applicable only in *small-step* SOS; this provides further motivation for avoiding the big-step style – at least for constructs whose operational semantics might conceivably involve abrupt termination.

3 MSOS

MSOS (Modular SOS) is a variant of SOS which dramatically improves modularity, at only very minor cost. Most of the features of SOS specifications, as reviewed in Section 2, carry over to MSOS. The differences are that in MSOS:

- configurations are *restricted* to abstract syntax and computed values,
- the labels are now the morphisms of a *category*, and
- adjacent labels in computations are required to be *composable*.

Surprisingly, this particular combination of the notions of LTS and category doesn't appear to have been previously exploited in connection with SOS.

3.1 Configurations

The specification of abstract syntax, computed values, auxiliary entities, and value-added syntax is exactly the same in MSOS as in SOS (see Tables 1, 2, 3, and 4).

The set Γ of configurations in MSOS is restricted to value-added syntax, and the set T of terminal configurations is restricted to computed values. Thus the specification of these sets for MSOS in Table 10 is actually superfluous, and could be left implicit.

Table 10
Configurations for MSOS

$$\Gamma = \mathrm{Exp} \cup \mathrm{Com} \cup \mathrm{Dec}$$
$$T = \mathbb{N} \cup \mathbb{T} \cup \{\mathbf{nil}\} \cup \mathrm{Env}$$

3.2 Generalized Transition Systems

In MSOS, as in SOS, the operational semantics of a programming language is modelled by a transition system (together with some notion of equivalence, see Section 4). The following kind of generalized transition system was introduced in [23]: [4]

Definition 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 LTTS.*

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

Notice that the transition system itself is *not* made into a category, since that would require the transition relation to be reflexive and transitive, both of which are inconsistent with the usual small-step style in MSOS.

Recall that a *category* [5] consists of:

- a set of objects O,
- a set of morphisms (also called arrows) A,
- functions 'source' and 'target' from A to O,
- a partial function from A^2 to A for composing morphisms, and
- a function from O to A giving an identity morphism for each object.

The functions above are required to satisfy some simple axioms, including associativity of composition, and unit properties for identity morphisms. We'll write compositions of morphisms α_1, α_2 in diagrammatic order: $\alpha_1 \,;\, \alpha_2$.

Proposition 3 *For each GTS $\langle \Gamma, \mathbb{A}, \longrightarrow, T \rangle$, an LTTS $\langle \Gamma^\bullet, A^\bullet, \longrightarrow^\bullet, T^\bullet \rangle$ can be constructed such that for each computation of the GTS, there is a computation of the LTTS with the same trace, and vice versa.*

[4] Generalized transition systems were called "arrow-labelled" in [23].
[5] Let us here disregard all foundational issues, such as the distinction between "small" and "large" categories.

A similar result is stated and proved in [22]. The construction is straightforward: each configuration of the LTTS is a pair consisting of a configuration of the LTTS and an object of the label category, and the transition relation is defined accordingly.

PROOF. Let O be the set of objects of $\mathbb{A}$, and A the set of morphisms. Define $\Gamma^\bullet = \Gamma \times O$, $T^\bullet = T \times O$, and $A^\bullet = A$. The construction is completed by letting $\langle\gamma, o\rangle \xrightarrow{\alpha}^\bullet \langle\gamma', o'\rangle$ hold in the LTTS iff $\gamma \xrightarrow{\alpha} \gamma'$ holds in the GTS, $\text{source}(\alpha) = o$, and $\text{target}(\alpha) = o'$.

With each (finite or infinite) GTS computation $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ we associate the LTTS computation $\langle\gamma, o\rangle \xrightarrow{\alpha_1}^\bullet \langle\gamma_1, o_1\rangle \xrightarrow{\alpha_2}^\bullet \ldots$, where $o = \text{source}(\alpha_1)$ and for $i \geq 1$, $o_i = \text{target}(\alpha_i) = \text{source}(\alpha_{i+1})$. If the computation in the GTS terminates with $\gamma_n \in T$, we always have $\langle\gamma_n, o_n\rangle \in T^\bullet$. The traces of the two computations are clearly the same.

Conversely, suppose that $\langle\gamma, o\rangle \xrightarrow{\alpha_1}^\bullet \langle\gamma_1, o_1\rangle \xrightarrow{\alpha_2}^\bullet \ldots$ is any (finite or infinite) computation in the defined LTTS. Then $o = \text{source}(\alpha_1)$ and for $i \geq 1$, $o_i = \text{target}(\alpha_i) = \text{source}(\alpha_{i+1})$. Hence α_i and α_{i+1} are composable for all $i \geq 1$. Moreover, if the computation in the LTTS terminates with $\langle\gamma_n, o_n\rangle \in T^\bullet$, we always have $\gamma_n \in T$. Hence $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ is a computation in the GTS, and the traces of the two computations are the same. $\square$

Note that the relationship between the GTS and the LTTS is stronger than that of an ordinary bisimulation, since the definition of computations in the GTS takes account of the composability of adjacent labels. For simplicity, we have defined $A^\bullet = A$, although this normally gives labels with some redundancy. In Section 3.9 we'll discuss how to obtain an SOS specification from an MSOS specification, and see how to eliminate all redundancy in the labels.

It is equally straightforward to go from an LTTS $\langle\Gamma, A, \longrightarrow, T\rangle$ to a corresponding GTS $\langle\Gamma, \mathbb{A}, \longrightarrow, T\rangle$, preserving computations: take $\mathbb{A}$ to be the free monoid A^* considered as a single-object category.

Proposition 4 *For each LTTS $\langle\Gamma, A, \longrightarrow, T\rangle$, a GTS $\langle\Gamma^\#, \mathbb{A}^\#, \longrightarrow^\#, T^\#\rangle$ can be constructed such that for each computation of the LTTS, there is a computation of the GTS with the same trace, and vice versa.*

PROOF. Define $\Gamma^\# = \Gamma$, $T^\# = T$, and let $\mathbb{A}^\#$ be the category given by the free monoid A^*. The construction is completed by letting $\gamma \xrightarrow{\alpha}^\# \gamma'$ hold in the GTS iff $\alpha \in A$ (considered as included in A^*) and $\gamma \xrightarrow{\alpha} \gamma'$ holds in the LTTS. Since all morphisms are composable in $\mathbb{A}^\#$, the computation of the LTTS is also a computation of the GTS, and has the same trace.

The converse direction holds due to the restriction of the transitions of the GTS to labels in A. $\square$

As already mentioned, the configurations in MSOS are restricted to value-added syntax. Auxiliary components of configurations in SOS, such as environments and stores, are not allowed in MSOS. There is only one place left for them to go: in the labels on transitions. The structure of the label category reflects how the information represented by the auxiliary components is supposed to "flow" when processed during computations.

3.3 Label Categories

Intuitively, a configuration in MSOS represents the part of the program which remains to be executed (and records already-computed values that are still needed), whereas the label on a transition represents all the information processed by the program in that step. Part of the label corresponds to the state of the processed information at the beginning of the transition, and part to its state at the end of the transition. For labels on adjacent transitions in a computation, the state at the end of the first transition must be identical to the state at the beginning of the second transition. Taking the states of processed information to be the objects of a category, labels obviously correspond to morphisms of the category.

Labels which are identity morphisms play a special rôle in MSOS: they indicate that transitions are inherently *unobservable*. Thus in contrast to SOS for process algebra, where the silent label τ is introduced ad hoc, MSOS provides a natural, built-in notion of unobservable transition. In Section 4, we'll exploit this fact in the definition of weak bisimulation for MSOS.

Apart from determining the states before and after a transition, a label in MSOS may also provide further information that corresponds directly to an ordinary label in SOS, and allows it to be distinguished from other labels between the same two states. There need be no correlation between this extra information in the labels on adjacent transitions in MSOS (as in SOS).

Let's see how all this works in some simple cases:

- Consider first labels that represent information which is processed like environments $\rho \in \mathrm{Env}$. Such information can be inspected by a transition, but not changed (since it is determined by the current context, and subsequent transitions have the same context). Thus the labels should be composable only when they are *identical*.

This is reflected by taking $\mathbb{A}$ to be a *discrete* category, where there is a single (identity) morphism for each object. The set of objects and the set of morphisms of $\mathbb{A}$ can both be identified with Env.

- Now for labels that represent information which is processed like stores $\sigma \in$ Store. Such information can be both inspected and changed by a transition. In principle, a single transition could change the values stored at all locations, so for each pair of stores σ and σ', there should be a label representing the change from σ to σ'. Two successive changes are composable only if the store left by the first change is the same as the store from which the second change starts.

  This is reflected by taking $\mathbb{A}$ to be a *preorder* category, where the set of objects is Store and the set of morphisms is Store^2, with the morphism $\langle \sigma, \sigma' \rangle$ going from σ to σ'. Composition eliminates intermediate stores. Identity morphisms are of the form $\langle \sigma, \sigma \rangle$.
- Finally, consider labels that represent information which is processed like observable actions or signals $a \in$ Act, together with a silent action τ. Although such information can be produced by a transition, it cannot be inspected by subsequent transitions. In principle, the action produced by one transition does not restrict the action produced by the next transition, so labels should always be composable.

  This is reflected by taking $\mathbb{A}$ to be a 1-object category, where the set of morphisms is Act*, the free monoid of sequences generated by Act. The identity morphism is the empty sequence, representing τ, and composition of morphisms is sequence concatenation. Single actions are represented by sequences of length one; longer sequences correspond to indivisible multi-action sequences.

It should be stressed that the above considerations concern only *sequences* of transitions. They do not affect the rules used to specify individual transitions. For instance, the environment used in a condition may well be different from that used in the conclusion (as will be illustrated below).

The three kinds of information processing considered above correspond to how environments, stores, and actions are usually treated in SOS. It appears that further kinds of information processing are not needed in practice (and even if they were, it is likely that they could be represented by appropriate choices of further basic label categories). What we do need, however, is to be able to combine them.

Since there are no general constraints relating environments, stores, and actions, it is appropriate to use a *product* of the three categories. The objects of the product category can be identified with pairs $\langle \rho, \sigma \rangle \in$ Env $\times$ Store (dropping the fixed component that corresponds to the single object of the monoid category). The morphisms, in contrast, correspond to quadruples $\langle \rho, \sigma, \sigma', t \rangle \in$ Env $\times$ Store $\times$ Store $\times$ Act* (where both σ and σ' come from the preorder category). Identity morphisms and composition of morphisms in the product category are determined by the component categories.

Taking the above product category as the label category, the transition $\gamma \xrightarrow{\langle\rho,\sigma,\sigma',t\rangle} \gamma'$ in MSOS corresponds exactly to the transition written $\rho \vdash \langle\gamma, \sigma\rangle \xrightarrow{t} \langle\gamma', \sigma'\rangle$ in SOS (provided that t is either the empty sequence or a single-action sequence).

More generally, we may take any number of instances of the three basic label categories defined above. For example, we could have separate environments for types and for ordinary values – or an extra store whose locations are used merely as unique identifiers for processes or channels.

It's possible to formalize the incremental construction of label categories as products using functors called label transformers [23]. However, bearing in mind Plotkin's initial aim with SOS regarding the use of "simple mathematics" [30], the following somewhat more syntactic approach seems preferable.

3.4 Label Components

Let RO, RW, and WO be disjoint sets of indices, and $I = RO \cup RW \cup WO$. In our examples, we'll use the meta-variables ρ, σ, and ε as indices, taking $RO = \{\rho\}$, $RW = \{\sigma\}$, and $WO = \{\varepsilon\}$, but in general, each set of indices may have any number of elements. Indices in RO are for "read-only" components, those in RW are for "read-write" components, and those in WO for "write-only" components.

Definition 5 *For each $i \in I$ let a set S_i be given, such that whenever $i \in WO$, S_i is a monoid. Each S_i determines a component category $\mathbb{A}_i$, as follows:*

- *if $i \in RO$, then $\mathbb{A}_i$ is the discrete category with S_i as its set of objects and also as its set of (identity) morphisms;*
- *if $i \in RW$, then $\mathbb{A}_i$ is the preorder category with S_i as its set of objects, and S_i^2 as its set of morphisms;*
- *if $i \in WO$, then $\mathbb{A}_i$ is the category with a single object, and with the monoid S_i as its set of morphisms.*

The label category defined by the sets S_i is their (indexed) product $\Pi_{i\in I}\mathbb{A}_i$.

Extending one of the subsets of I with a fresh index i for some given set S_i corresponds to applying a functor (called a fundamental label transformer in [23]) that adds a new component to the product category $\mathbb{A}$. Moving an index between the subsets of I can also be useful: moving it from RO to RW causes a discrete component category to be replaced by a preorder category, and moving it from *WO* to *RW* causes a monoid component category to be replaced by a preorder category (other moves are not needed).

The crucial point, however, is not so much how to construct label categories where the labels have particular components, but rather to provide a clear notation for referring to (and possibly replacing) some components of a label *without any mention at all of what other components that label might or might not have.* The original notation (with *set* and *get* operations) proposed for this purpose [23] was somewhat clumsy, and required explicit mention of label transformers. The notation introduced below allows labels in MSOS rules to be specified much more concisely and perspicuously, and the systematic treatment of primed and unprimed indices allows the construction of the label category to be left completely implicit.

The idea is to use an unprimed index i to refer to a component that can be "read" at the start of a transition, and a primed index i' to refer to a component that can be "written" at the end of a transition. For instance, $\alpha.\rho$ and $\alpha.\sigma$ refer to the current environment and store, and $\alpha.\sigma'$ refers to the updated store. Notice that the label component α_σ is the pair $\langle\alpha.\sigma, \alpha.\sigma'\rangle$.

Definition 6 *Let α be a morphism of the product category $\Pi_{i\in I}\mathbb{A}_i$, and $i \in I$. The notation $\alpha.i$ and $\alpha.i'$ is defined in terms of the morphism component α_i as follows:*

- *if $i \in RO$, define $\alpha.i$ to be α_i ($\alpha.i'$ is undefined);*
- *if $i \in RW$ and $\alpha_i = \langle s_1, s_2\rangle$, define $\alpha.i$ to be s_1, and $\alpha.i'$ to be s_2;*
- *if $i \in WO$, define $\alpha.i'$ to be α_i ($\alpha.i$ is undefined).*

The label category $\mathbb{A}$ for an MSOS is specified by declaring the indices (such as ρ, σ, σ') used for referring to components, together with a corresponding set for each component. The structure of labels corresponds to that of records in Standard ML, so to specify our label components, let us adopt a notation similar to that used there for record types, as illustrated in Table 11 below. The ellision '. . . ' indicates that further components of labels may be specified elsewhere. When the same index is declared both primed and unprimed, the same set has to be associated with it each time. Moreover, the sets associated with indices that are used only primed should always be monoids.

Table 11
Label components

$$A = \{\rho : \mathrm{Env}; \sigma, \sigma' : \mathrm{Store}, \ldots\}$$

We adopt also Standard ML notation for individual label *patterns* in rules. For example:

Table 12
MSOS rules

$$\frac{e_0 \xrightarrow{X} e_0'}{e_0 \ bop \ e_1 \xrightarrow{X} e_0' \ bop \ e_1} \tag{18}$$

$$\frac{e_1 \xrightarrow{X} e_1'}{con_0 \ bop \ e_1 \xrightarrow{X} con_0 \ bop \ e_1'} \tag{19}$$

$$\frac{bop = +, \quad n = n_0 + n_1}{n_0 \ bop \ n_1 \longrightarrow n} \tag{20}$$

$$\frac{\mathrm{unobs}\{\rho, \ldots\}, \quad \rho(x) = con}{x \xrightarrow{\{\rho,\ldots\}} con} \tag{21}$$

- $\{\rho{=}\rho_0, \ldots\}$ specifies labels α such that $\alpha.\rho = \rho_0$;
- $\{\sigma{=}\sigma_0, \sigma'{=}\sigma_1, \ldots\}$ specifies labels α such that $\alpha.\sigma = \sigma_0$ and $\alpha.\sigma' = \sigma_1$;
- $\{\sigma, \ldots\}$ abbreviates $\{\sigma{=}\sigma, \ldots\}$, allowing the meta-variable σ to refer directly to the σ-component of the label.

The explicit '. . .' in the above notation for labels is obligatory, and ensures that unmentioned components of labels are never excluded. Different occurrences of '. . .' in the same rule stand for the *same* set of unmentioned components; the symbol '. . . ' may be regarded formally as a meta-variable ranging over parts of labels. [6] The order in which components of labels are written is, of course, insignificant. (Note that Plotkin used a similar notation in his notes for expressing finite functions such as environments and stores.)

3.5 Rules

Rules in MSOS are written exactly the same way as in SOS. The meta-variables X and U have a fixed usage in MSOS: X ranges over arbitrary labels in A, whereas U is restricted to labels that are identity morphisms, which are used to label "unobservable" transitions. We may abbreviate $\gamma \xrightarrow{U} \gamma'$ to $\gamma \longrightarrow \gamma'$ when we don't need to refer to U elsewhere in the same rule. The condition $\mathrm{unobs}(X)$ expresses directly that X is unobservable.

Table 12 shows how the rules from Table 6 (which were reformulated in Table 8 when adding stores) are specified in MSOS.

[6] The notation $\{\rho{=}\rho_0|X\}$, analogous to Prolog's notation for list patterns, might be misread as set comprehension. When required, we may distinguish different occurrences of '. . . ' by subscripts.

Table 13
MSOS rules for declarations

$$\frac{d \xrightarrow{X} d'}{\mathbf{let}\, d\, \mathbf{in}\, e \xrightarrow{X} \mathbf{let}\, d'\, \mathbf{in}\, e} \tag{22}$$

$$\frac{e \xrightarrow{\{\rho=\rho_1[\rho_0],\ldots\}} e'}{\mathbf{let}\, \rho_0\, \mathbf{in}\, e \xrightarrow{\{\rho=\rho_1,\ldots\}} \mathbf{let}\, \rho_0\, \mathbf{in}\, e'} \tag{23}$$

$$\mathbf{let}\, \rho_0\, \mathbf{in}\, \mathit{con} \longrightarrow \mathit{con} \tag{24}$$

$$\frac{e \xrightarrow{X} e'}{\mathbf{const}\, x = e \xrightarrow{X} \mathbf{const}\, x = e'} \tag{25}$$

$$\mathbf{const}\, x = \mathit{con} \longrightarrow (x \mapsto \mathit{con}) \tag{26}$$

$$\frac{d_0 \xrightarrow{X} d_0'}{d_0 \,;\, d_1 \xrightarrow{X} d_0' \,;\, d_1} \tag{27}$$

$$\frac{d_1 \xrightarrow{\{\rho=\rho_1[\rho_0],\ldots\}} d_1'}{\rho_0 \,;\, d_1 \xrightarrow{\{\rho=\rho_1,\ldots\}} \rho_0 \,;\, d_1'} \tag{28}$$

$$\rho_0 \,;\, \rho_1 \longrightarrow \rho_0[\rho_1] \tag{29}$$

In (18) and (19), the use of the same label variable X in both the condition and the conclusion of the rule ensures that transitions for the subexpression have the same environment, initial store, and final store as the corresponding transitions for the enclosing expression – and similarly for any further components that labels might have. All this comes *automatically*, without any explicit mention of environments or stores in the rules.

In (20) and (21), the labels on the transitions are restricted to identity morphisms. This is just what is needed to prevent side-effects, such as store changes or observable actions, in what are supposed to be unobservable transitions.

The formulation of the rules specified in Table 12 is independent of whether or not labels have stores as components. In fact (18)–(20) are independent of *all* label components, and only (21) requires labels to have a ρ-component.

Table 13 gives MSOS rules for let-expressions and for three kinds of declarations; the corresponding SOS rules are given in the appendix.

The main rules of interest are (23) and (28). They illustrate how the ρ-component of a label can be adjusted to reflect its extension with a computed environment ρ_0, which represents the bindings due to local declarations. The ρ-component of the label on the transition in the conclusion has value ρ_1, whereas that of the label on the transition in the condition is $\rho_1[\rho_0]$ (recall the notation for overriding introduced in Section 2.3). The unmentioned components of the two labels are required to be equal, but otherwise unconstrained.

The formulation of the rules specified in Table 13 is independent of whether or not labels have stores as components. In fact the rules for elaborating constant declarations to environments are even independent of the presence of the ρ-component.

Table 14
MSOS rules for commands

$$\frac{c_0 \xrightarrow{X} c_0'}{c_0\,;c_1 \xrightarrow{X} c_0'\,;c_1} \tag{30}$$

$$\mathbf{nil}\,;c_1 \longrightarrow c_1 \tag{31}$$

$$\frac{d \xrightarrow{X} d'}{d\,;c \xrightarrow{X} d'\,;c} \tag{32}$$

$$\frac{c \xrightarrow{\{\rho=\rho_1[\rho_0],\ldots\}} c'}{\rho_0\,;c \xrightarrow{\{\rho=\rho_1,\ldots\}} \rho_0\,;c'} \tag{33}$$

$$\rho_0\,;\mathbf{nil} \longrightarrow \mathbf{nil} \tag{34}$$

$$\frac{e \xrightarrow{X} e'}{\mathbf{if}\,e\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1 \xrightarrow{X} \mathbf{if}\,e'\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1} \tag{35}$$

$$\mathbf{if}\,\mathrm{tt}\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1 \longrightarrow c_0 \tag{36}$$

$$\mathbf{if}\,\mathrm{ff}\,\mathbf{then}\,c_0\,\mathbf{else}\,c_1 \longrightarrow c_1 \tag{37}$$

$$\mathbf{while}\,e\,\mathbf{do}\,c \longrightarrow \mathbf{if}\,e\,\mathbf{then}\,c\,;\mathbf{while}\,e\,\mathbf{do}\,c\,\mathbf{else}\,\mathbf{nil} \tag{38}$$

Table 14 gives MSOS rules for command sequences, local declarations, and the familiar if- and while-commands; the corresponding SOS rules for all these constructs can be found in the appendix.

Plotkin treated **nil** as a command taking one step to execute, whereas here we have treated it as a computed value (and hence as a final configuration). The examples given in the appendix show that it would be quite straightforward to reformulate the examples in Plotkin's notes to follow the style illustrated in Table 14; but how about the other way, can we match Plotkin's treatment of commands in MSOS?

To do this, we would need to remove **nil** from the set of computed values, and introduce a new value, say **null**, for commands to compute:

$$\begin{aligned}\Gamma &= \mathrm{Exp} \cup \mathrm{Com} \cup \mathrm{Dec} \\ T &= \mathbb{N} \cup \mathbb{T} \cup \{\mathbf{null}\} \cup \mathrm{Env}\end{aligned}$$

It isn't necessary to include **null** in value-added syntax for commands, since the rules will never require the insertion of this value in place of a command. Thus the meta-variable c doesn't range over **null**.

The corresponding rules for **nil** and command sequencing would then be as follows (similar changes would be needed in the rules for all constructs that involve commands):

$$\mathbf{nil} \longrightarrow \mathbf{null} \tag{39}$$

$$\frac{c_0 \xrightarrow{X} c_0'}{c_0\,;c_1 \xrightarrow{X} c_0'\,;c_1} \tag{40}$$

$$\frac{c_0 \xrightarrow{X} \mathbf{null}}{c_0\,;c_1 \xrightarrow{X} c_1} \tag{41}$$

As can be seen, the rules given in Table 14 are a bit simpler than the ones that follow Plotkin's style. However, the latter generally give rise to fewer unobservable transitions (at least when atomic commands other than **nil** are introduced), so they can be considered more "economical" in that sense. In any case, the MSOS framework allows both styles, leaving the choice to the specifier. (This admittedly undermines the idea of having a single "definitive" MSOS for all constructs; more on this issue in Section 3.8.)

Some further differences from Plotkin's notes occur in connection with if- and while-commands, where he used the (reflexive and) transitive closure $\longrightarrow^*$ of the small-step transition relation for expressions to get the effect of the big-step style. Here, however, we'll not illustrate how one could reformulate the MSOS rules given in Table 14 to match his rules more closely in that respect,

Table 15
MSOS rules for variables

$$\frac{\mathrm{unobs}\{\rho,\sigma,\ldots\},\quad \rho(x)=l,\quad \sigma(l)=con}{x \xrightarrow{\{\rho,\sigma,\ldots\}} con} \tag{42}$$

$$\frac{e \xrightarrow{X} e'}{x := e \xrightarrow{X} x := e'} \tag{43}$$

$$\frac{\mathrm{unobs}\{\rho,\sigma{=}\sigma_0,\sigma'{=}\sigma_0,\ldots\},\quad \rho(x)=l}{x := con \xrightarrow{\{\rho,\sigma=\sigma_0,\sigma'=\sigma_0[l\mapsto con],\ldots\}} \mathbf{nil}} \tag{44}$$

$$\frac{e \xrightarrow{X} e'}{\mathbf{var}\, x := e \xrightarrow{X} \mathbf{var}\, x := e'} \tag{45}$$

$$\frac{\mathrm{unobs}\{\sigma{=}\sigma_0,\sigma'{=}\sigma_0,\ldots\},\quad l\notin\mathrm{dom}(\sigma)}{\mathbf{var}\, x := con \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[l\mapsto con],\ldots\}} (x\mapsto l)} \tag{46}$$

since we have reasons to reserve the big-step style for constructs which are essentially mathematical, without possibilities of side-effects, divergence, or abrupt termination – and this is not the case for expressions, in general.

Table 15 gives MSOS rules for variable dereferencing, assignment, and declaration; again, the corresponding SOS rules can be found in the appendix. Notice that (42) gives a transition only when the ρ- and σ-components of the label are such that $\rho(x) \in \mathrm{Loc}$, and the value stored at the location l given by $\rho(x)$ is in the set Con. Similarly, (44) and (46) give transitions only when the value *con* is in the set of storable values SVal.

3.6 Styles

MSOS, like SOS, allows both the small- and big-step styles. Table 16 shows how the rules from Table 6 would look in the big-step style:

Label composition $X_1;X_2$ is often used explicitly in the big-step style of MSOS, so as to combine the labels for sub-computations. This has the pleasant side-effect of showing the intended order of the sub-computations. Moreover, the use of label composition in MSOS entirely subsumes the "store convention" that was adopted in the Definition of Standard ML [17]: when labels have σ- and σ'-components, the composability of X_1 and X_2 ensures that the σ'-component of X_1 must be the same as the σ-component of X_2.

It appears to be possible to provide further ways of composing labels (al-

Table 16
Modular natural semantics rules

$$\frac{e_0 \xrightarrow{X_1} n_0, \quad e_1 \xrightarrow{X_2} n_1}{e_0 + e_1 \xrightarrow{(X_1;X_2)} n_0 + n_1} \tag{47}$$

$$con \longrightarrow con \tag{48}$$

$$\frac{\text{unobs}\{\rho, \ldots\}, \quad \rho(x) = con}{x \xrightarrow{\{\rho,\ldots\}} con} \tag{49}$$

Table 17
Modular static semantics rules

$$\frac{e_0 \longrightarrow \tau, \quad e_1 \longrightarrow \tau}{e_0 = e_1 \longrightarrow \mathbf{bool}} \tag{50}$$

$$n \longrightarrow \mathbf{int}, \quad t \longrightarrow \mathbf{bool} \tag{51}$$

$$\frac{\text{unobs}\{\alpha, \ldots\}, \quad \alpha(x) = \tau}{x \xrightarrow{\{\alpha,\ldots\}} \tau} \tag{52}$$

though MSOS doesn't provide any notational support for this at present). For instance, suppose that the labels of a big-step MSOS are arbitrary (finite) sequences of the usual labels. A label category can be defined by taking composition to be sequence concatenation. This label category can then be enriched with a relation corresponding to arbitrary "shuffling" of sequences. By replacing $X_1 ; X_2$ in (47) by a label required to be a shuffle of X_1 and X_2, we get a big-step MSOS rule for arbitrary interleaving. The labels on big-step computations of entire programs should then be restricted to composable sequences of the usual labels, reflecting that no further interleaving is possible.

The possibility of specifying interleaving in a big-step MSOS is a technical curiosity, but of little practical relevance for applications of MSOS, which generally stick to the small-step style. One important case where the big-step style definitely seems to be preferable is for specifying *static semantics*, including type-checking. Typically, a conventional SOS for static semantics involves environments, but neither stores nor labels on transitions. Thus in a corresponding MSOS, all labels would be identity morphisms. Since such labels are composable iff they are equal, there is no need to use label composition explicitly in big-step rules for static semantics. In fact, recalling that we allow $\gamma \longrightarrow \gamma'$ as an abbreviation for $\gamma \xrightarrow{U} \gamma'$ when there's no need to refer to U directly in a rule, we can simply omit the labels altogether in most rules. Table 17 gives a simple illustration of MSOS rules for type-checking, assuming that the α-component of a label provides a typing environment mapping variables to their types.

3.7 Abrupt Termination

Finally, let us consider the MSOS of constructs that may terminate "abruptly", due to errors, exceptions, breaks, or goto's. The standard style for specifying dynamic errors in conventional SOS, as illustrated in Section 2.9, is to add "error rules" that allow error configurations to propagate through each construct. Such propagation rules are quite tedious, especially in big-step SOS; for the Definition of Standard ML, an "exception convention" was introduced so that all the corresponding exception propagation rules could be left implicit.

In connection with MSOS, a modular and elegant technique for specifying both abrupt termination and its handling has been discovered (B. Klin, personal communication, October 2001). The basic idea is to make abrupt termination *observable in the labels* on transitions that give rise to it. The closest enclosing handler for the abrupt termination can then monitor each transition of its body, and terminate it immediately when the label indicates that (the right kind of) abrupt termination is possible. (Such monitoring is reminiscent of synchronization between concurrent processes, although here, the handler is synchronizing with its own sub-construct.)

Let's illustrate the new technique in connection with the simple dynamic errors from Plotkin's notes. We assume that in abstract syntax, complete programs are now distinguished from commands; other forms of abrupt termination would involve further syntactic constructs, such as exception handlers.

We introduce a fresh "write-only" label component $\varepsilon' : \{err\}^?$, where $\{err\}^?$ is the two-element monoid $\{err, ()\}$ such that the unit is $()$ and multiplication involving err always results in err.[7] We also introduce an extra configuration, written **stuck** (although it is never inspected in the rules, and it doesn't matter whether there are any transitions from it or not) and include it in the value-added syntax of expressions.

The rules are given in Table 18. When $n_0 < n_1$, rule (53) gives rise to a transition where the ε-component of the label is set to the value err. Rule (54) checks that the ε-component of the label is $()$, indicating that no error has occurred, so the execution of the program can continue. Rule (55), in contrast, notices immediately when the ε-component of the label is different from $()$, and discards the command configuration c' so as to terminate the entire program.

[7] The free monoid $\{err\}^*$ could be used instead of $\{err\}^?$, but the extra elements aren't of any use here.

Table 18
MSOS of dynamic errors

Abstract syntax:

$$p ::= \textbf{program}\, c$$

Configurations:

$$e ::= \textbf{stuck}$$
$$\Gamma \;=\; \ldots \cup \{\textbf{stuck}\}$$

Labels:

$$A \;=\; \{\varepsilon' : \{err\}^?, \ldots\}$$

$$\frac{bop = -, \quad n_0 < n_1, \quad \text{unobs}\{\varepsilon' = (\,), \ldots\}}{n_0 \; bop \; n_1 \xrightarrow{\{\varepsilon' = err, \ldots\}} \textbf{stuck}} \tag{53}$$

$$\frac{c \xrightarrow{\{\varepsilon', \ldots\}} c', \quad \varepsilon' = (\,)}{\textbf{program}\, c \xrightarrow{\{\varepsilon', \ldots\}} \textbf{program}\, c'} \tag{54}$$

$$\frac{c \xrightarrow{\{\varepsilon', \ldots\}} c', \quad \varepsilon' \neq (\,)}{\textbf{program}\, c \xrightarrow{\{\varepsilon', \ldots\}} \textbf{nil}} \tag{55}$$

$$\textbf{program nil} \longrightarrow \textbf{nil} \tag{56}$$

No further rules are needed for propagating errors through other constructs: the ε-component of the label is always propagated – along with any other unmentioned components – by the *normal* MSOS rules for compound constructs. Moreover, rules that give rise to *normal* transitions for atomic constructs necessarily specify labels whose unmentioned components are identity morphisms, so in particular, their ε-components are automatically the unit $(\,)$.

The MSOS description of exception-handling (and similar constructs involving abrupt termination) is equally straightforward. If the smallest enclosing handler matches the raised exception – which is used as the ε-component of the label instead of err – the handler replaces its body by the appropriate code, and reflects that the computation is now proceeding normally again by setting the ε-component of the label on the transition to $(\,)$. If the handler doesn't match the exception, or if there is no exception, the transition has the same label as that for the body of the handler [28]. The description of the 'finally' construct (as found in Java) is only slightly more complicated: in the case that the body raised an exception, one has to append a statement to re-raise the same exception at the (normal) end of the handling code.

The above technique was developed in connection with MSOS, but it could also be used in a conventional small-step SOS of a programming language, as illustrated in the appendix. The only problem is that it would usually require reformulation of all the usual rules for constructs: to add labels on transitions, or ensure propagation of extra components of existing labels. With MSOS, in contrast, the rules for compound constructs *always* propagate unmentioned observable components of labels, and adding an extra component to labels doesn't require the reformulation of any rules at all.

It seems unlikely that an analogous technique could be provided for use in big-step MSOS. Thus the small-step style has a distinct advantage for the specification of constructs which might involve abrupt termination.

3.8 Modularity

As shown in Table 8, allowing the inspection of stored values in expressions requires reformulation of the conventional SOS rules given in Table 6. The corresponding MSOS rules were given in Table 12; these are definitive, and never[8] need reformulating. All that may be needed when adding new constructs to a language described in MSOS is to extend the label components. For example, we might have started with ρ : Env as the only component, then added σ : Store, and finally σ' : Store as well. (The exact comparison between SOS and MSOS is here slightly obscured by the fact that, for simplicity, we are using the same label category in MSOS for the transitions of all syntactic constructs.)

As we observed in Section 3.5 when comparing our MSOS rules for commands to those in Plotkin's notes, MSOS still has some degrees of freedom, such as whether or not to use computed values directly in abstract syntax, as well as the more fundamental choice between the small- and big-step styles. Even the label category components can be varied, e.g., by using the preorder category for an "imperative" treatment of environments. Further variations may be required in connection with non-standard operational semantic models, for instance introducing locations for actions in process algebra.

Thus we cannot expect that MSOS will lead to *canonical* specifications of operational semantics. Nevertheless, MSOS specifications in the general style illustrated here do exhibit considerable modularity, and can indeed be easily extended, as well as reused in descriptions of many different programming languages. Further evidence of the high degree of modularity that can be obtained using MSOS is provided by the descriptions of action notation [25,27],

[8] It is however an open problem to give an MSOS for continuation-handling constructs such as call/cc.

the core of Concurrent ML [26], and various constructs from Standard ML covered in lecture courses at Aarhus.

3.9 Relationship Between MSOS and SOS Rules

Suppose that we have fixed a set of configurations Γ, a label category $\mathbb{A}$, and a set of final configurations T. A set of MSOS rules then defines the transition relation $\longrightarrow$ of a GTS. By Proposition 3, an LTTS $\langle \Gamma^\bullet, A^\bullet, \longrightarrow^\bullet, T^\bullet \rangle$ can be constructed from the GTS such that for each computation of the GTS, there is a computation of the LTTS with the same trace, and vice versa.

A corresponding construction can be applied to the MSOS specification, mapping it to a SOS specification which defines (essentially) the same LTTS as the one constructed from the GTS specified by the MSOS. Let's consider the construction of each part of the SOS specification in turn:

- The specifications of abstract syntax, computed values, auxiliary entities, and value-added syntax in the SOS are exactly the same as in the MSOS.
- The SOS specification of the sets of configurations $\Gamma^\bullet$ and $T^\bullet$ is determined by the MSOS specifications of Γ, T, and $\mathbb{A}$: $\Gamma^\bullet = \Gamma \times S_1 \times \cdots \times S_n$ and $T^\bullet = T \times S_1 \times \cdots \times S_n$, where the sets S_j are those associated with indices of read-only and read-write label components.
- The SOS specification of the set of labels $A^\bullet$ is determined by the MSOS specification of $\mathbb{A}$: $A^\bullet = S'_1 \times \cdots \times S'_m$, where the sets S'_j are the monoids associated with indices of write-only components. Notice that the LTTS construction in Proposition 3 corresponds to taking $A^\bullet = A$; here, we omit components of $A^\bullet$ which are already included in the configurations $\Gamma^\bullet$, since they are redundant.
- An SOS rule is constructed systematically from each MSOS rule, as follows: First, all occurrences of the meta-variables X and U in the MSOS rule are eliminated in favour of record patterns with meta-variables corresponding to their specified components (occurrences of U give rise to double uses of the same meta-variable for read-write components, and to constants denoting the units of the monoids for write-only components). Moreover, occurrences of '. . . ' in record patterns are replaced by any missing fields that were specified as components of $\mathbb{A}$. Uses of $X.i$ and $U.i$ are replaced by the selected components.

  Then each transition $t \xrightarrow{\{i_1=t_1,\cdots,i_n=t_n\}} t'$ in the MSOS rule is converted to an SOS transition of the form $\ldots \vdash \langle t, \ldots \rangle \xrightarrow{\cdots} \langle t', \ldots \rangle$, where the precise locations of the terms t_i in the SOS notation are determined by the kind and ordering of the indices i_j: read-only components go to the left of the '$\vdash$', the unprimed and primed versions of read-write components go to the left, resp. right of the '$\longrightarrow$', and the write-only components go above the '$\longrightarrow$'.

Finally, any remaining uses of record patterns in side-conditions are replaced by conjunctions of side-conditions not involving record patterns.

The appendix shows the result of applying the above construction to the collected illustrative examples of MSOS given throughout this chapter (some further transformations were applied to substitute for variables defined by side-conditions).

A formal presentation of the construction of SOS rules from MSOS rules, and a proof of its correctness, are left to future work.

4 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.

4.1 Strong Bisimulation

An MSOS defines a generalized transition system $GTS = \langle \Gamma, \mathbb{A}, \longrightarrow, T \rangle$ with an underlying labelled terminal transition system $LTTS = \langle \Gamma, A, \longrightarrow, T \rangle$, 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 [15], adjusted to take account of terminal configurations in LTTS:

Definition 7 *Let $LTTS = \langle \Gamma, A, \longrightarrow, T \rangle$ be a labelled terminal 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 γ_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 γ_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 $\gamma_1 = \gamma_2$.*

γ_1, γ_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 LTTS to GTS, and the usual proof techniques are available. Since the configurations γ 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 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 doesn't 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 [23] was based on the reduction from GTS to LTTS, 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.)

4.2 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, but it's not clear that the extra bother of doing that is worthwhile. For a general notion of equivalence, it's 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 τ. 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 τ to our labels. Moreover, definitions of weak bisimulation don't depend on τ 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 [15] is formulated for MSOS as follows (branching and other varieties of bisimulation would be defined analogously):

Definition 8 *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 $\gamma_1 \stackrel{\alpha}{\Longrightarrow} \gamma_1'$ then for some γ_2', $\gamma_2 \stackrel{\hat{\alpha}}{\Longrightarrow} \gamma_2'$ and $\langle \gamma_1', \gamma_2' \rangle \in R$;*
- *whenever $\gamma_2 \stackrel{\alpha}{\Longrightarrow} \gamma_2'$ then for some γ_1', $\gamma_1 \stackrel{\hat{\alpha}}{\Longrightarrow} \gamma_1'$ 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:

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

5 Related Work

The modular approach to SOS presented here, MSOS, was inspired by the Moggi's monad transformers [18], and in particular by Liang and Hudak's practical development of a modular monadic semantics framework [14]. As mentioned in the Introduction, the search for modularity in SOS was stimulated by Wansbrough and Hamer's [40] modular monadic semantics of much of the action notation used in action semantics, the original SOS definition of which [19] lacks modularity. MSOS attempts to transfer the practical benefits of monad transformers from denotational to operational semantics. However, this has been achieved only for simple monad transformers concerned with incorporating new components of the processed information, since the flow of control in MSOS is generally expressed by the patterns of transitions in the rules (as in conventional SOS) and is not affected by the components of labels. The illustrated technique for the modular treatment of abrupt termination in MSOS was discovered by Klin, and doesn't appear to be closely related to the monad transformer for exceptions.

The basic ideas of MSOS were first explored by the author in [21]. The technique of incorporating all semantic information in labels has previously been proposed as a general principle for SOS also by Degano and Priami [8], and exploited by them to obtain parametricity in the framework of Enhanced Operational Semantics. However, they did not abstract from the structure of labels (which is a crucial step for obtaining full modularity and extensibility), nor did they consider partial composition of labels. The Tile Model framework of Gadducci and Montanari [11] provides categorical structure on labels, but is otherwise not closely related to MSOS.

There has been extensive work on various formats of small-step SOS (see [10] for references), but the conservativity results obtained there concern extensions with new syntax and rules, rather than changes to labels. An SOS format with terms as labels has been proposed by Bernstein [4], but modularity was not considered. The work of Turi and Plotkin [38] on the fusion of denotational and operational semantics doesn't appear to address modularity either.

A non-structural but quite succinct approach to operational semantics is to give an (unlabelled) reduction semantics for applications of *evaluation contexts* $C[t]$, following Felleisen et al. [9,42]. The use of evaluation contexts appears to provide some inherent modularity, but obtaining full modularity may involve the introduction of many artificial internal steps [6]. Reppy's evaluation-context semantics for ML concurrency primitives [34,35] has better modularity than the SOS given in [5] – see [26] for a detailed comparison of it with an MSOS for the same language. See also [24] for a more general survey of frameworks for logical specification of operational semantics.

6 Conclusion

In this paper, we have reviewed the conventional SOS framework, and defined MSOS as a variant of SOS where configurations are restricted to abstract syntax and computed values, the labels are the morphisms of a category, and adjacent labels in computations are required to be composable. We have provided a simple and modular way of defining label categories, based on the indices used to refer to the components of labels. And we have introduced an efficient notation for referring to and replacing particular components of labels without mentioning other components. All this allows the MSOS rules for individual language constructs to be completely independent of each other, and encourages the development of a library of MSOS modules that can be reused – without any reformulation – in the descriptions of different programming languages.

The illustrative examples given in Section 3 are comparable with examples in Plotkin's notes, and systematically related to the SOS examples given in the appendix below. We have also shown how abrupt termination can be described in a completely modular way, avoiding the usual tedious propagation rules, in both SOS and MSOS – provided that the style is the usual small-step one. Finally, we have given straightforward definitions of both strong and weak bisimulation for MSOS (the latter exploiting the built-in distinction of unobservable labels) and related MSOS to other work.

Much work remains ahead, regarding both theoretical and pragmatic aspects of MSOS. On the theoretical side, further investigation of bisimulation and other equivalences for MSOS is needed, and it should be investigated whether MSOS rules can be restricted to a format which would ensure that bisimulation is a congruence. A satisfactory way of describing continuation-passing constructs (such as call/cc) in either SOS or MSOS has yet to be found.

For practical use of MSOS to specify the operational semantics of entire programming languages, it is important to establish an electronic library of reusable modules giving the MSOS rules for all commonly-occurring constructs; this entails the development of a basic, language-independent abstract syntax, the precise design of which is non-trivial.

Proper tool support needs to be developed too: the author presently validates MSOS rules by transcribing them to Prolog (preserving modularity), but this transcription should be automated, and some checks on the well-formedness of the MSOS rules should be implemented – especially in connection with use of MSOS in courses on formal semantics at the undergraduate level. The Prolog code corresponding to the MSOS rules specified in Section 3 is available at `http://www.brics.dk/~pdm/JLAP-MSOS.pl`. It includes a logic grammar, allowing programs in the described language to be parsed, as well as further code that facilitates inspection of intermediate states of computations.

Readers who are interested in working on the particular topics mentioned above, or in making use of MSOS to describe full-scale programming languages, are kindly requested to contact the author.

Acknowledgements

Thanks to the anonymous referees, who made helpful suggestions regarding both the content and the organization of the paper. Special thanks to Luca Aceto and Wan Fokkink as editors for advice, encouragement, and patience; and to Gordon Plotkin for developing SOS and for writing such inspiring and insightful lecture notes during his visit to Aarhus in 1981. Re-reading those notes while on sabbatical at SRI International, Menlo Park, 1998-9, stimulated the development of MSOS, and discussions in that period with José Meseguer and others helped to clarify the essential categorical notions associated with labels in MSOS.

The author is supported by BRICS (Basic Research in Computer Science), funded by the Danish National Research Foundation.

A Illustrative Examples of (Non-Modular) SOS

This appendix shows how the MSOS rules given in Section 3.5 can be formulated in SOS. See Section 3.9 for discussion of how to obtain SOS rules systematically from MSOS rules, so as to obtain (essentially) the same computations.

The style of the rules given here differs in several respects from that of the rules given in Plotkin's notes. In particular, our rules are consistently small-step, let commands compute **nil**, and exploit the novel treatment of errors explained in Section 3.7.

Abstract Syntax See Table 1, extended by:

Programs: $p ::= \textbf{program}\, c$

Computed Values See Table 2.

Auxiliary Entities See Table 3.

Value-Added Syntax

Expressions: $e ::= \textbf{stuck}$
Declarations: $d ::= \rho$

Configurations

$$\Gamma = (\text{Exp} \cup \text{Com} \cup \text{Dec} \cup \text{Prog}) \times \text{Env} \times \text{Store}$$
$$T = (\mathbb{N} \cup \mathbb{T} \cup \{\textbf{nil}\} \cup \text{Env}) \times \text{Env} \times \text{Store}$$

Labels

$\varepsilon \in A = \{(\,), \mathit{err}\}$ $\quad$ ($\xrightarrow{(\,)}$ is abbreviated to $\longrightarrow$)

Expression Rules

$$\frac{\rho \vdash \langle e_0, \sigma \rangle \xrightarrow{\varepsilon} \langle e_0', \sigma' \rangle}{\rho \vdash \langle e_0\ bop\ e_1, \sigma \rangle \xrightarrow{\varepsilon} \langle e_0'\ bop\ e_1, \sigma' \rangle} \tag{A.1}$$

$$\frac{\rho \vdash \langle e_1, \sigma \rangle \xrightarrow{\varepsilon} \langle e_1', \sigma' \rangle}{\rho \vdash \langle con_0\ bop\ e_1, \sigma \rangle \xrightarrow{\varepsilon} \langle con_0\ bop\ e_1', \sigma' \rangle} \tag{A.2}$$

$$\frac{bop = +, \quad n = n_0 + n_1}{\rho \vdash \langle n_0\ bop\ n_1, \sigma \rangle \longrightarrow \langle n, \sigma \rangle} \tag{A.3}$$

$$\frac{bop = -, \quad n_0 < n_1}{\rho \vdash \langle n_0\ bop\ n_1, \sigma \rangle \xrightarrow{err} \langle \textbf{stuck}, \sigma \rangle} \tag{A.4}$$

$$\frac{\rho(x) = con}{\rho \vdash \langle x, \sigma \rangle \longrightarrow \langle con, \sigma \rangle} \tag{A.5}$$

$$\frac{\rho(x) = l, \quad \sigma(l) = con}{\rho \vdash \langle x, \sigma \rangle \longrightarrow \langle con, \sigma \rangle} \tag{A.6}$$

$$\frac{\rho \vdash \langle d, \sigma\rangle \xrightarrow{\varepsilon} \langle d', \sigma'\rangle}{\rho \vdash \langle \mathbf{let}\, d\, \mathbf{in}\, e, \sigma\rangle \xrightarrow{\varepsilon} \langle \mathbf{let}\, d'\, \mathbf{in}\, e, \sigma'\rangle} \tag{A.7}$$

$$\frac{\rho[\rho_0] \vdash \langle e, \sigma\rangle \xrightarrow{\varepsilon} \langle e', \sigma'\rangle}{\rho \vdash \langle \mathbf{let}\, \rho_0\, \mathbf{in}\, e, \sigma\rangle \xrightarrow{\varepsilon} \langle \mathbf{let}\, \rho_0\, \mathbf{in}\, e', \sigma'\rangle} \tag{A.8}$$

$$\rho \vdash \langle \mathbf{let}\, \rho_0\, \mathbf{in}\, con, \sigma\rangle \longrightarrow \langle con, \sigma\rangle \tag{A.9}$$

Command Rules

$$\frac{\rho \vdash \langle c_0, \sigma\rangle \xrightarrow{\varepsilon} \langle c_0', \sigma'\rangle}{\rho \vdash \langle c_0\, ;\, c_1, \sigma\rangle \xrightarrow{\varepsilon} \langle c_0'\, ;\, c_1, \sigma'\rangle} \tag{A.10}$$

$$\rho \vdash \langle \mathbf{nil}\, ;\, c_1, \sigma\rangle \longrightarrow \langle c_1, \sigma\rangle \tag{A.11}$$

$$\frac{\rho \vdash \langle d, \sigma\rangle \xrightarrow{\varepsilon} \langle d', \sigma'\rangle}{\rho \vdash \langle d\, ;\, c, \sigma\rangle \xrightarrow{\varepsilon} \langle d'\, ;\, c, \sigma'\rangle} \tag{A.12}$$

$$\frac{\rho[\rho_0] \vdash \langle c, \sigma\rangle \xrightarrow{\varepsilon} \langle c', \sigma'\rangle}{\rho \vdash \langle \rho_0\, ;\, c, \sigma\rangle \xrightarrow{\varepsilon} \langle \rho_0\, ;\, c', \sigma'\rangle} \tag{A.13}$$

$$\rho \vdash \langle \rho_0\, ;\, \mathbf{nil}, \sigma\rangle \longrightarrow \langle \mathbf{nil}, \sigma\rangle \tag{A.14}$$

$$\frac{\rho \vdash \langle e, \sigma\rangle \xrightarrow{\varepsilon} \langle e', \sigma'\rangle}{\rho \vdash \langle \mathbf{if}\, e\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \xrightarrow{\varepsilon} \langle \mathbf{if}\, e'\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma'\rangle} \tag{A.15}$$

$$\rho \vdash \langle \mathbf{if}\, \mathrm{tt}\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \longrightarrow \langle c_0, \sigma\rangle \tag{A.16}$$

$$\rho \vdash \langle \mathbf{if}\, \mathrm{ff}\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \longrightarrow \langle c_1, \sigma\rangle \tag{A.17}$$

$$\rho \vdash \langle \mathbf{while}\, e\, \mathbf{do}\, c, \sigma\rangle \longrightarrow \langle \mathbf{if}\, e\, \mathbf{then}\, c\, ;\, \mathbf{while}\, e\, \mathbf{do}\, c\, \mathbf{else}\, \mathbf{nil}, \sigma\rangle \tag{A.18}$$

$$\frac{\rho \vdash \langle e, \sigma \rangle \xrightarrow{\varepsilon} \langle e', \sigma' \rangle}{\rho \vdash \langle x := e, \sigma \rangle \xrightarrow{\varepsilon} \langle x := e', \sigma' \rangle} \tag{A.19}$$

$$\frac{\rho(x) = l}{\rho \vdash \langle x := con, \sigma \rangle \longrightarrow \langle \mathbf{nil}, \sigma[l \mapsto con] \rangle} \tag{A.20}$$

Declaration Rules

$$\frac{\rho \vdash \langle e, \sigma \rangle \xrightarrow{\varepsilon} \langle e', \sigma' \rangle}{\rho \vdash \langle \mathbf{const}\, x = e, \sigma \rangle \xrightarrow{\varepsilon} \langle \mathbf{const}\, x = e', \sigma' \rangle} \tag{A.21}$$

$$\rho \vdash \langle \mathbf{const}\, x = con, \sigma \rangle \longrightarrow \langle (x \mapsto con), \sigma \rangle \tag{A.22}$$

$$\frac{\rho \vdash \langle e, \sigma \rangle \xrightarrow{\varepsilon} \langle e', \sigma' \rangle}{\rho \vdash \langle \mathbf{var}\, x := e, \sigma \rangle \xrightarrow{\varepsilon} \langle \mathbf{var}\, x := e', \sigma' \rangle} \tag{A.23}$$

$$\frac{l \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle \mathbf{var}\, x := con, \sigma \rangle \longrightarrow \langle (x \mapsto l), \sigma[l \mapsto con] \rangle} \tag{A.24}$$

$$\frac{\rho \vdash \langle d_0, \sigma \rangle \xrightarrow{\varepsilon} \langle d_0', \sigma' \rangle}{\rho \vdash \langle d_0 \,;\, d_1, \sigma \rangle \xrightarrow{\varepsilon} \langle d_0' \,;\, d_1, \sigma' \rangle} \tag{A.25}$$

$$\frac{\rho[\rho_0] \vdash \langle d_1, \sigma \rangle \xrightarrow{\varepsilon} \langle d_1', \sigma' \rangle}{\rho \vdash \langle \rho_0 \,;\, d_1, \sigma \rangle \xrightarrow{\varepsilon} \langle \rho_0 \,;\, d_1', \sigma' \rangle} \tag{A.26}$$

$$\rho \vdash \langle \rho_0 \,;\, \rho_1, \sigma \rangle \longrightarrow \langle \rho_0[\rho_1], \sigma \rangle \tag{A.27}$$

Program Rules

$$\frac{\rho \vdash \langle c, \sigma \rangle \longrightarrow \langle c', \sigma' \rangle}{\rho \vdash \langle \mathbf{program}\, c', \sigma \rangle \longrightarrow \langle \mathbf{program}\, c', \sigma' \rangle} \tag{A.28}$$

$$\frac{\rho \vdash \langle c, \sigma \rangle \xrightarrow{err} \langle c', \sigma' \rangle}{\rho \vdash \langle \mathbf{program}\, c, \sigma \rangle \xrightarrow{err} \langle \mathbf{nil}, \sigma' \rangle} \tag{A.29}$$

$$\rho \vdash \langle \mathbf{program}\, \mathbf{nil}, \sigma \rangle \longrightarrow \langle \mathbf{nil}, \sigma \rangle \tag{A.30}$$

References

[1] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 3, pages 197–292. Elsevier Science, 2001.

[2] 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, 1991.

[3] E. Astesiano, M. Bidoit, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Comput. Sci.*, 286(2):153–196, 2002.

[4] K. L. Bernstein. A congruence theorem for structured operational semantics of higher-order languages. In *Proc. LICS'98*, pages 153–163. IEEE, 1998.

[5] 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.

[6] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In M. Hagiya and J. C. Mitchell, editors, *TACS'94, Symposium on Theoretical Aspects of Computer Software*, LNCS 789, pages 244–272. Springer, 1994.

[7] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS, IFIP Series. Springer, 2004. To appear.

[8] P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, 1996.

[9] 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, 1986*, pages 193–217. North-Holland, 1987.

[10] W. J. Fokkink and C. Verhoef. A conservative look at operational semantics with variable binding. *Information and Computation*, 146(1):24–54, 1998.

[11] F. Gadducci and U. Montanari. The tile model. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 133–166. The MIT Press, 2000.

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

[13] G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, LNCS 247, pages 22–39. Springer, 1987.

[14] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96, Proc. 6th European Symposium on Programming*, LNCS 1058, pages 219–234. Springer, 1996.

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

[16] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

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

[18] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.

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

[20] P. D. Mosses. Theory and practice of action semantics. In *MFCS'96 Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland*, LNCS 1113, pages 37–61. Springer, 1996.

[21] P. D. Mosses. Semantics, modularity, and rewriting logic. In *WRLA'98, Proc. 2nd Int. Workshop on Rewriting Logic and its Applications*, ENTCS 15, 1998.

[22] P. D. Mosses. Foundations of modular SOS. Technical report, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version of [23].

[23] 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*, LNCS 1672, pages 70–80. Springer, 1999.

[24] P. D. Mosses. Logical specification of operational semantics. In *CSL'99, Proc. Conf. on Computer Science Logic*, LNCS 1683, pages 32–49. Springer, 1999.

[25] P. D. Mosses. A modular SOS for Action Notation. Technical report, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version of [?].

[26] P. D. Mosses. A modular SOS for ML concurrency primitives. Technical report, Dept. of Computer Science, Univ. of Aarhus, 1999.

[27] P. D. Mosses. AN-2: Revised action notation—syntax and semantics. Available at `http://www.brics.dk/~pdm/papers/Mosses-AN-2-Semantics/`, 2000.

[28] P. D. Mosses. Pragmatics of modular SOS. In A. M. Haeberer, editor, *AMAST'02, Proc. 9th International Conference on Algebraic Methods and Software Technology*, LNCS 2422, pages 21–40. Springer, 2002.

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

[30] G. D. Plotkin. A structural approach to operational semantics. Technical report, Dept. of Computer Science, Univ. of Aarhus, 1981. Reprinted in JLAP, special issue on SOS, 2004.

[31] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In F. Honsell and M. Miculan, editors, *FOSSACS 2001, Foundations of Software Science and Computation Structures*, LNCS 2030, pages 1–24. Springer, 2001.

[32] G. D. Plotkin and J. Power. Semantics for algebraic operations (extended abstract). In S. Brookes and M. Mislove, editors, *Proc. MFPS XVII*, ENTCS 45. Elsevier, 2001.

[33] G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *FOSSACS 2002, Foundations of Software Science and Computation Structures*, LNCS 2303, pages 342–356. Springer, 2002.

[34] 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.

[35] J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Computer Science Dept., Cornell Univ., 1992. Tech. Rep. TR 92-1285.

[36] J. C. Reynolds. Using category theory to define implicit coercions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, LNCS 94, pages 211–258. Springer, 1980.

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

[38] D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *Proc. LICS'97.* IEEE, 1997.

[39] K. Wansbrough. A modular monadic action semantics. Master's thesis, Dept. of Computer Science, Univ. of Auckland, 1997.

[40] K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997.

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

[42] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Dept. of Computer Science, Rice University, 1991.

Contents

1	Introduction	1
1.1	Background	2
1.2	Overview	3
2	Conventional SOS	3
2.1	Syntax	4
2.2	Computed Values	5
2.3	Auxiliary Entities	6
2.4	Configurations for SOS	8
2.5	Transition Systems for SOS	9
2.6	Rules in SOS	10
2.7	Styles of SOS	12
2.8	Modularity in SOS	13
2.9	Abrupt Termination	15
3	MSOS	16
3.1	Configurations	16
3.2	Generalized Transition Systems	17
3.3	Label Categories	19
3.4	Label Components	21
3.5	Rules	23
3.6	Styles	27
3.7	Abrupt Termination	29
3.8	Modularity	31
3.9	Relationship Between MSOS and SOS Rules	32
4	Equivalence in MSOS	33

4.1 Strong Bisimulation 33

4.2 Weak Bisimulation 35

5 Related Work 36

6 Conclusion 37

Acknowledgements 38

A Illustrative Examples of (Non-Modular) SOS 38

References 42

Recent BRICS Report Series Publications

RS-05-7 **Peter D. Mosses.** ***Modular Structural Operational Semantics*****. February 2005. 46 pp. Appears in *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.**

RS-05-6 **Karl Krukow and Andrew Twigg.** ***Distributed Approximation of Fixed-Points in Trust Structures*****. February 2005. 41 pp.**

RS-05-5 **A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations.** ***Dariusz Biernacki and Olivier Danvy and Kevin Millikin*****. February 2005.**

RS-05-4 **Andrzej Filinski and Henning Korsholm Rohde.** ***Denotational Aspects of Untyped Normalization by Evaluation*****. February 2005.**

RS-05-3 **Olivier Danvy and Mayer Goldberg.** ***There and Back Again*****. January 2005. iii+16 pp. Extended version of an article to appear in *Fundamenta Informatica*. This version supersedes BRICS RS-02-12.**

RS-05-2 **Dariusz Biernacki and Olivier Danvy.** ***On the Dynamic Extent of Delimited Continuations*****. January 2005. ii+30 pp.**

RS-05-1 **Mayer Goldberg.** ***On the Recursive Enumerability of Fixed-Point Combinators*****. January 2005. 7 pp. Superseedes BRICS report RS-04-25.**

RS-04-41 **Olivier Danvy.** ***Sur un Exemple de Patrick Greussay*****. December 2004. 14 pp.**

RS-04-40 **Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde.** ***Fast Partial Evaluation of Pattern Matching in Strings*****. December 2004. 22 pp. To appear in TOPLAS. Supersedes BRICS report RS-03-20.**

RS-04-39 **Olivier Danvy and Lasse R. Nielsen.** ***CPS Transformation of Beta-Redexes*****. December 2004. ii+11 pp. Superseedes an article to appear in *Information Processing Letters* and BRICS report RS-00-35.**

RS-04-38 **Olin Shivers and Mitchell Wand.** ***Bottom-Up β-Substitution: Uplinks and λ-DAGs*****. December 2004.**

RS-04-37 **Jørgen Iversen and Peter D. Mosses.** ***Constructive Action Semantics for Core ML*****. December 2004. 68 pp. To appear in a special *Language Definitions and Tool Generation* issue of the journal *IEE Proceedings Software*.**