Basic Research in Computer Science

BRICS RS-99-54 P. D. Mosses: Foundations of Modular SOS

Foundations of Modular SOS

Peter D. Mosses

BRICS Report Series **RS-99-54**

ISSN 0909-0878 **December 1999**

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

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

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/99/54/`

Foundations of Modular SOS*

Peter D. Mosses[1]

BRICS and Department of Computer Science,
University of Aarhus, Denmark

Abstract. A novel form of labelled transition system is proposed, where the labels are the arrows of a category, and adjacent labels in computations are required to be composable. Such transition systems provide the foundations for modular SOS descriptions of programming languages. Three fundamental ways of transforming label categories, analogous to monad transformers, are provided, and it is shown that their applications preserve computations in modular SOS. The approach is illustrated with fragments taken from a modular SOS for ML concurrency primitives.

1 Introduction

SOS (structural operational semantics) is a widely-used framework for defining process algebras [17, e.g.] and programming languages [18, e.g.]. Following Plotkin [30], SOS has often been preferred to the more abstract framework of denotational semantics. The labelled transition systems that provide the foundations for SOS are themselves well-studied mathematical objects, with major applications in software (and hardware) engineering.

Modular SOS is a form of SOS that ensures a high degree of *modularity*: the transition rules for each construct are completely independent of the presence or absence of other constructs in the described language. When one extends or changes the described language, the description can be extended or changed accordingly, without reformulation—even though new kinds of information processing may be required. This is in marked contrast to conventional SOS, where modularity tends to be quite poor: when extending a pure functional language with concurrency primitives and/or references, for instance, the original specification of the transition rules has to be completely reformulated [4].

In denotational semantics, the problem of obtaining good modularity has received much attention, and has to a large extent been solved by introducing so-called monad transformers. Modular SOS provides an analogous solution for operational semantics.

The basic idea of Modular SOS is to incorporate *all* semantic entities as components of labels. Thus configurations are restricted to syntax and computed values. The foundations of Modular SOS involve a novel form of labelled transition system (LTS), where *the labels are the arrows of a category*. In contrast

* Full version of [24], reporting research carried out while visiting SRI International and Stanford University, USA

to other frameworks where labels are equipped with categorical structure (e.g. Tile Logic [11] and Rewriting Logic [15]), composition here is generally a *partial* operation, and computations are restricted to those where all adjacent labels are composable. Note that the labels are no longer the simple atomic actions often used in studies of process algebra, but here usually have entities such as environments and stores as components; so do the *objects* of the label category, which correspond to the *states* of the processed information.

Any arrow-labelled transition system (ALTS) can be reduced to an ordinary LTS, and the usual notions of derivative and bisimilarity lifted accordingly; a version of higher-order bisimulation may also be defined directly.

Three fundamental *label transformers* have been identified; they preserve the computations specified by a modular SOS, and their order of application is irrelevant. The label transformers are analogous to some simple monad transformers. The one which transforms the label category to incorporate new *context* information (such as the current environment) adds the same sort of component both to arrows and to objects, and composition preserves the value of that component. Also the transformer which incorporates *mutable* information (such as the current store) adds a corresponding component to each object, whereas it extends each arrow with a pair of such components; composition on pairs is as for binary relations. Finally, the transformer which incorporates *emitted* information (such as synchronization signals) adds a corresponding component to each arrow, but leaves the objects essentially unchanged.

Plan of the Paper: Section 2 starts by recalling the basic notions of SOS and LTS. Section 3 defines what an ALTS is, and shows how any ALTS can be reduced to a corresponding ordinary LTS. Section 4 provides some simple illustrations of label categories. Section 5 defines the three fundamental ways of transforming label categories to incorporate further kinds of processed information. Section 6 gives some illustrative excerpts from a modular SOS of ML concurrency primitives. Section 7 discusses the relationship of Modular SOS to other work. Section 8 concludes by indicating what remains to be done. An extended abstract of this paper, omitting proofs and some other details, is available [24].

2 Conventional SOS

In the conventional SOS framework [30, 31] programs (and all their constituent phrases) are generally modelled as *labelled transition systems*:

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

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

The main characteristic feature of SOS is that transitions are specified inductively, according to the syntactic structure of the described language, by *rules*:

$$\frac{\gamma_1 \xrightarrow{\alpha_1} \gamma_1' \quad \dots \quad \gamma_n \xrightarrow{\alpha_n} \gamma_n'}{\gamma \xrightarrow{\alpha} \gamma'}.$$

The syntactic components of $\gamma_1, \dots, \gamma_n$ are generally sub-phrases of the syntactic component of γ. Other formulae, such as equations, may be used as *side-conditions* on rules (often listed together with the premises). The intended transition relation is the least relation that is closed under the given rules.[1]

There are two distinct styles of SOS: in so-called *small-step SOS*, each transition in a computation generally corresponds to an indivisible item of information processing, such as adding two computed numbers, or assigning a computed number to a variable; in *big-step SOS*, also known as *Natural Semantics* [13], a computation is a single transition leading directly to a terminal configuration, corresponding to the combination of many items of information processing. The two styles may be mixed in the same description, e.g., big-step for expression evaluation and small-step for command execution; alternatively, the transitive closure of a small-step transition relation can be used to represent a big-step relation [30].

Intermediate configurations in small-step SOS generally involve an extension of abstract syntax, allowing any phrase to be replaced by its computed value. Let us refer to such an extended syntax as *value-added.* The pure abstract syntax can be defined as the initial algebra in a class of algebras; the value-added syntax corresponds to the algebra freely generated by the sets of values, one for each syntactic category. (In some languages, the computed values can be identified with canonical terms of the original syntax, so no extension is needed.)

Configurations often involve familiar semantic components, such as stores that map variables to their assigned values. Environments (mapping identifiers to their denoted values) are however usually treated as separate arguments of a *relative* transition relation $\rho \vdash \gamma \xrightarrow{\alpha} \gamma'$ [13, 30]; this complication can be avoided by using syntactic substitution instead of environments (although it is quite tedious to define substitution when binding constructs introduce local scopes for variables). Input, output, and synchronization signals are all generally recorded in the labels on transitions.

Detailed explanations of the conventional SOS framework are available in the literature [2, 12, 13, 29–32, 35]. The lack of modularity in conventional SOS is evident in many papers, for instance [4].

3 Modular SOS

Modular SOS (MSOS) is a particularly simple and uniform style of SOS. The essential idea is to use the *labels* on transitions to represent general *information*

[1] A more complicated definition is needed when negations of assertions of transitions are allowed in premises [9].

processing steps; the configurations merely keep track of the flow of control and computed values, and are therefore restricted to syntax and computed values (i.e., value-added syntax, see Sect. 2). The transition relation is required to be ternary ($\gamma \xrightarrow{\alpha} \gamma'$), so the only place left for the usual semantic components of transitions (such as environments and stores) is in the labels.

In a transition $\gamma \xrightarrow{\alpha} \gamma'$, the label α must itself determine the *state* of the processed information both before and after the step. Two such transitions can be adjacent in a computation only when the state after the first and the state before the second are identical. This intuition is conveniently represented by regarding the labels as the *arrows of a category*, with the states as the objects of the category. The foundations for MSOS are provided by such *arrow-labelled transition systems*. (Surprisingly, this appears to be a novel combination of the familiar notions of LTS and category.)

Note that MSOS does not require any knowledge of Category Theory. All that is needed is the basic concept of a category, which may here be regarded as a loosely-specified class of partial algebras, corresponding to an abstract datatype:

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

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

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

3.1 Arrow-Labelled Transition Systems

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

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

Identity arrows are regarded as *silent* or *unobservable*; they generally label transitions that merely reduce the configuration without changing the state, e.g., computing a new value from already-computed arguments, or propagating an exception. In an ALTS corresponding to a process algebra such as CCS [16], there is only one object in the label category, and its identity arrow corresponds to the unobservable action (τ).

It is straightforward to generalize the usual inductive definition of the transitive closure of the transition relation to ALTS:

Definition 4. *Let $(\Gamma, T, \mathbb{A}, \longrightarrow)$ be an ALTS; then $(\Gamma, T, \mathbb{A}, \longrightarrow^{+})$ is an ALTS, where $\longrightarrow^{+}$ is the least relation such that:*

$$\frac{\gamma \xrightarrow{\alpha_1}{}^{+} \gamma_1 \quad \gamma_1 \xrightarrow{\alpha_2}{}^{+} \gamma_2 \quad \alpha = \alpha_1 \mathbin{;} \alpha_2}{\gamma \xrightarrow{\alpha}{}^{+} \gamma_2} \qquad \frac{\gamma \xrightarrow{\alpha} \gamma'}{\gamma \xrightarrow{\alpha}{}^{+} \gamma'}.$$

Notice that $\gamma \xrightarrow{\alpha}{}^{+} \gamma' \in T$ iff there exists a finite computation $\gamma \xrightarrow{\alpha_1} \gamma_1 \ldots \xrightarrow{\alpha_n} \gamma_n = \gamma'$ such that $\alpha = \alpha_1 \mathbin{;} \cdots \mathbin{;} \alpha_n$.

3.2 Reduction of ALTS to LTS

It is straightforward to reduce any ALTS to a corresponding LTS: just augment the configurations $\gamma \in \Gamma$ with the states $o \in |\mathbb{A}|$ of the label category $\mathbb{A}$, and forget the categorical structure of $\mathbb{A}$.

Definition 5. *Let $(\Gamma, T, \mathbb{A}, \longrightarrow)$ be an ALTS. Let the LTS $(\Gamma^{\bullet}, T^{\bullet}, \mathbb{A}^{\bullet}, \longrightarrow^{\bullet})$ be defined by taking $\Gamma^{\bullet} = \Gamma \times |\mathbb{A}|$, $T^{\bullet} = T \times |\mathbb{A}|$, $\mathbb{A}^{\bullet} = \mathbb{A}$, and for all $\gamma, \gamma' \in \Gamma$, $o, o' \in |\mathbb{A}|$ and $\alpha \in \mathbb{A}$, $(\gamma, o) \xrightarrow{\alpha}{}^{\bullet} (\gamma', o')$ iff $\gamma \xrightarrow{\alpha} \gamma'$ and $\alpha : o \to o'$.*

Proposition 1. *There is a 1-1 correspondence between the computations of $(\Gamma, T, \mathbb{A}, \longrightarrow)$ and those of $(\Gamma^{\bullet}, T^{\bullet}, \mathbb{A}^{\bullet}, \longrightarrow^{\bullet})$.*

Proof. For each computation $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ of the ALTS there is the computation $(\gamma, o) \xrightarrow{\alpha_1}{}^{\bullet} (\gamma_1, o_1) \xrightarrow{\alpha_2}{}^{\bullet} \ldots$ where $o = \mathit{pre}(\alpha_1)$ and for $i \geq 1$, $\mathit{post}(\alpha_i) = o_i = \mathit{pre}(\alpha_{i+1})$. Conversely, suppose that $(\gamma, o) \xrightarrow{\alpha_1}{}^{\bullet} (\gamma_1, o_1) \xrightarrow{\alpha_2}{}^{\bullet} \ldots$ is a computation of the LTS; then for $i \geq 1$, $\mathit{post}(\alpha_i) = o_i = \mathit{pre}(\alpha_{i+1})$, hence $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ is a computation of the ALTS. $\square$

Notice that taking $\mathbb{A}^{\bullet} = \mathbb{A}$ normally gives labels with some redundancy, since the source and target of the label on any transition are determined by the configurations before and after the transition. Such redundancy is harmless, but it could be eliminated if desired.

Any LTS $(\Gamma, T, \mathbb{A}, \longrightarrow)$ can be made into an ALTS $(\Gamma, T, \mathbb{A}', \longrightarrow')$ by taking $\mathbb{A}' = \mathbb{A}^*$ (the one-object category corresponding to the free monoid on $\mathbb{A}$) and letting $\gamma_1 \xrightarrow{\alpha}{}' \gamma_2$ iff $\gamma_1 \xrightarrow{\alpha} \gamma_2$ for $\alpha \in \mathbb{A}$. The computations of the ALTS correspond exactly to those of the original LTS. The constructed ALTS reduces to an LTS which, when the (redundant) labels in $\mathbb{A}^* \setminus \mathbb{A}$ are removed, is isomorphic to the original ALTS.

3.3 Bisimilarity

Let us first recall the usual notion of bisimulation for ordinary LTS [16], adjusted to take account of terminal configurations:

Definition 6. *Let $(\Gamma, T, \mathbb{A}, \longrightarrow)$ be an LTS. $S \subseteq \Gamma \times \Gamma$ is a* strong bisimulation *iff $(\gamma_1, \gamma_2) \in S$ implies, for all $\alpha \in \mathbb{A}$,*

- *whenever $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ then for some γ_2', $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ and $(\gamma_1', \gamma_2') \in S$;*
- *whenever $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ then for some γ_1', $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ and $(\gamma_1', \gamma_2') \in S$; and*
- *whenever $\gamma_1 \in T$ or $\gamma_2 \in T$ then $\gamma_1 = \gamma_2$.*

γ_1, γ_2 are strongly bisimilar*, $\gamma_1 \sim \gamma_2$, iff $(\gamma_1, \gamma_2) \in S$ for some strong bisimulation S.*

Thus $\sim$ is the largest strong bisimulation.

The reduction from ALTS to LTS given above induces a definition of bisimulation for ALTS:

Definition 7. *Let $(\Gamma, T, \mathbb{A}, \longrightarrow)$ be an ALTS. Then $S \subseteq (\Gamma \times |\mathbb{A}|) \times (\Gamma \times |\mathbb{A}|)$ is a* strong arrow-labelled bisimulation *for the ALTS iff S is a strong bisimulation for the corresponding LTS $(\Gamma^\bullet, T^\bullet, \mathbb{A}^\bullet, \longrightarrow^\bullet)$.*

Configurations γ_1 and γ_2 are said to be strongly arrow-labelled bisimilar *when there exists a strong arrow-labelled bisimulation S such that for all $o \in |\mathbb{A}|$, $((\gamma_1, o), (\gamma_2, o)) \in S$.*

When $|\mathbb{A}|$ is trivial (a singleton), the notion of strong arrow-labelled bisimulation defined for ALTS corresponds exactly to the usual notion. The need to consider pairs of configurations and states reflects that the labels on adjacent transitions in computations are required to be composable.

The following generalization of strong arrow-labelled bisimulation for ALTS corresponds to a notion of higher-order bisimulation, allowing for a subsidiary relation on labels:

Definition 8. *Let $(\Gamma, T, \mathbb{A}, \longrightarrow)$ be an ALTS. A pair of relations $S \subseteq (\Gamma \times |\mathbb{A}|) \times (\Gamma \times |\mathbb{A}|)$, $\tilde{S} \subseteq \mathbb{A} \times \mathbb{A}$ is called a* strong higher-order arrow-labelled bisimulation *iff $((\gamma_1, o_1), (\gamma_2, o_2)) \in S$ implies that:*

- *whenever $\gamma_1 \xrightarrow{\alpha_1} \gamma_1'$ with $\alpha_1 : o_1 \to o_1'$, then $\gamma_2 \xrightarrow{\alpha_2} \gamma_2'$ for some $\alpha_2, \gamma_2', o_2'$ with $\alpha_2 : o_2 \to o_2'$, $((\gamma_1', o_1'), (\gamma_2', o_2')) \in S$, and $(\alpha_1, \alpha_2) \in \tilde{S}$;*
- *whenever $\gamma_2 \xrightarrow{\alpha_2} \gamma_2'$ with $\alpha_2 : o_2 \to o_2'$, then $\gamma_1 \xrightarrow{\alpha_1} \gamma_1'$ for some $\alpha_1, \gamma_1', o_1'$ with $\alpha_1 : o_1 \to o_1'$, $((\gamma_1', o_1'), (\gamma_2', o_2')) \in S$, and $(\alpha_1, \alpha_2) \in \tilde{S}$; and*
- *$\gamma_1 \in T$ iff $\gamma_2 \in T$.*

Notice that terminal configurations are no longer required to be identical, and that whether they are in a bisimulation relation may also depend on the states of the ALTS.

With MSOS, higher-order bisimulation appears to be needed not only for dealing with equivalence of higher-order concurrent languages, but also for non-concurrent languages where functions can be computed and assigned to variables (or, when using environments rather than substitution, bound to variables).

Notions of weak bisimulation for arrow-labelled transition systems can be defined similarly, exploiting that transitions labelled by identity arrows are inherently unobservable.

4 Basic Label Categories

Let us consider some simple examples of label categories. In the next section they are generalized to generic *transformers* of label categories.

Definition 9. *The category* ***TrivCat*** *is a category with a single object and a single (identity) arrow.*

Taking labels in ***TrivCat*** gives an ALTS corresponding to an unlabelled transition system where the configurations have no semantic components at all: the transition relation $e \longrightarrow e'$ is essentially pure term rewriting (except that it is not in general closed with respect to contexts).

Definition 10. *Let* $\textsc{Env}$ *be a set of environments (i.e., finite maps from identifiers to values). Then* ***Discrete***$(\textsc{Env})$ *is the discrete category with the environments* $\rho \in \textsc{Env}$ *both as the objects and as the only (identity) arrows. Composition of two arrows is defined only when they are the same environment:* $\rho_1 \,\mathbf{;}\, \rho_2 = \rho$ *iff* $\rho_1 = \rho_2 = \rho$.

Taking labels in ***Discrete***$(\textsc{Env})$ gives an ALTS corresponding to an LTS with an (unlabelled) *relative* transition relation $\rho \vdash e \longrightarrow e'$ (the precise definition of which was left as an exercise in [30]).

Definition 11. *Let* $\textsc{Store}$ *be a set of stores (i.e., finite maps from addresses to values). Then* ***Pairs***$(\textsc{Store})$ *is the category with stores* $s \in \textsc{Store}$ *as objects, and with the* pairs *of stores* (s, s') *being the only arrows between the objects* s, s'. *Identity arrows are of the form* (s, s). *Arrow composition* $(s_1, s_1') \,\mathbf{;}\, (s_2, s_2')$ *is defined by* $(s_1, s_1') \,\mathbf{;}\, (s_2, s_2') = (s_1, s_2')$ *iff* $s_1' = s_2$.

Taking labels in ***Pairs***$(\textsc{Store})$ gives an ALTS corresponding to an (unlabelled) LTS where configurations are of the form (e, s).

Definition 12. *Let* $\textsc{Act}$ *be some set of actions. Then* ***Monoid***$(\textsc{Act}^*, \mathit{concat}, [\,])$ *is the category corresponding to the free monoid generated by* $\textsc{Act}$. *This category has a single object, and the arrows are finite sequences* $a_1 \ldots a_n$ *of elements* $a_i \in \textsc{Act}$. *Composition is totally defined as sequence concatenation, concat; the empty sequence* $[\,]$ *is the identity arrow.*

Taking labels in ***Monoid***$(\textsc{Act}^*, \mathit{concat}, [\,])$ gives an ALTS corresponding to an LTS where the labels are just single actions $a \in \textsc{Act}$, together with the unobservable action τ, as is usual in studies of process algebra [17, e.g.].

5 Fundamental Label Transformers

The label categories defined in Sect. 4 correspond to fundamentally different ways of processing information: allowing it to be *inspected*, or to be both *inspected and changed*, or merely to be *provided*. The fundamental label transformers defined below add such information processing to arbitrary label categories.

Label categories provide some auxiliary notation that allows the various components of the labels to be inspected and changed independently of each other. To avoid dependence on the order in which transformers are composed, particular components are referred to via symbolic indices $i \in \textsc{Index}$. All components of labels are taken from some universe $\textsc{Univ}$. The operations

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

are completely undefined when $\mathbb{A}$ is $\boldsymbol{\mathit{TrivCat}}$.

Definition 13. *Let $\mathbb{B}$ be a category, and $i \in \textsc{Index}$. Then the label transformer $\boldsymbol{\mathit{LabTrans}}(i, \mathbb{B})$ maps any label category $\mathbb{A}$ to $\mathbb{A} \times \mathbb{B}$, and extends the operations get and set from $\mathbb{A}$ to $\mathbb{A} \times \mathbb{B}$ by defining*

$$\mathit{get}((\alpha, u), j) = \begin{cases} u, & \mathit{if}\ i = j \\ \mathit{get}(\alpha, j), & \mathit{otherwise} \end{cases}$$

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

For any $\mathbb{A}$, $\mathbb{B}$ the projection from $\mathbb{A} \times \mathbb{B}$ to $\mathbb{A}$ is a functor. Moreover, for any object $b \in |\mathbb{B}|$ the embedding that maps objects $a \in |\mathbb{A}|$ to (a, b) and arrows $\alpha \in \mathbb{A}$ to $(\alpha, \mathit{id}(b))$ is also a functor. Thus a label transformer $F : \mathbb{A} \to \mathbb{A}'$ defined in terms of $\boldsymbol{\mathit{LabTrans}}$ may always be regarded as as projection functor together with a family of embedding functors, with embedding followed by projection being the identity functor on $\mathbb{A}$.

Definition 14. *$\boldsymbol{\mathit{ContextInfo}}(i, E)$ is $\boldsymbol{\mathit{LabTrans}}(i, \boldsymbol{\mathit{Discrete}}(E))$.*

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

Definition 15. *$\boldsymbol{\mathit{MutableInfo}}(i, S)$ is $\boldsymbol{\mathit{LabTrans}}(i, \boldsymbol{\mathit{Pairs}}(S))$.*

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

Definition 16. *$\boldsymbol{\mathit{EmittedInfo}}(i, A, f, \tau)$ is $\boldsymbol{\mathit{LabTrans}}(i, \boldsymbol{\mathit{Monoid}}(A, f, \tau))$.*

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

The above label transformers appear to be adequate for constructing appropriate label categories for use in MSOS descriptions of constructs of conventional programming languages (such as those described in conventional SOS in [30]) as well as for those of less conventional languages such as Concurrent ML [28] and Action Notation [26]). Notice that they are all concerned with the flow of information rather than that of control, the latter being expressed directly through transition rules in SOS-based frameworks.

A crucial property of the fundamental label transformers is that they preserve the computations specified by a set of transition rules. Thus to extend an MSOS one may first apply a label transformer—essentially without changing the semantics of those constructs that have already been described—and then proceed to exploit the new component of labels in the description of new constructs. This is different from the conservative extension properties generally found in the literature on LTS [10, e.g.], where one considers adding new configurations and rules, leaving the labels on transitions unchanged.

Proposition 2. *Let sets of configurations Γ, T be given. Let $\mathbb{A}$, $\mathbb{A}'$ be label categories related by functors $F : \mathbb{A} \to \mathbb{A}'$, $G : \mathbb{A}' \to \mathbb{A}$. Let R be a set of positive transition rules, such that the holding of side-conditions is preserved by F and G, and let $\longrightarrow$, $\longrightarrow'$ be the transition relations specified by R with labels ranging over $\mathbb{A}$, respectively $\mathbb{A}'$.*

Then for each computation $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ in $(\Gamma, T, \mathbb{A}, \longrightarrow)$ there is a corresponding computation $\gamma \xrightarrow{\alpha_1'}{}' \gamma_1 \xrightarrow{\alpha_2'}{}' \ldots$ in $(\Gamma, T, \mathbb{A}', \longrightarrow')$, and vice versa.

Proof. For any $\gamma, \gamma_1 \in \Gamma$, $\alpha_1 \in \mathbb{A}$, there is a transition $\gamma \xrightarrow{\alpha_1} \gamma_1$ iff there is a proof tree for it formed from the rules in R (with axioms as leaves), instantiating all variables by elements of the sets over which they range, and satisfying all the side-conditions of the rules. Replacing each element $\alpha \in \mathbb{A}$ in the tree by $F(\alpha)$ yields a proof tree for $\gamma \xrightarrow{F(\alpha_1)}{}' \gamma_1$, since the holding of side-conditions on rules is assumed to be preserved by F.

Moreover, if $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \gamma_2$, then $\alpha_1 \,;\, \alpha_2$ is defined, and the functoriality of F gives the definedness of $F(\alpha_1) \,;\, F(\alpha_2)$, hence $\gamma \xrightarrow{F(\alpha_1)}{}' \gamma_1 \xrightarrow{F(\alpha_2)}{}' \gamma_2$. By induction, we get that for each (finite or infinite) computation $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ in $(\Gamma, T, \mathbb{A}, \longrightarrow)$ there is the corresponding computation $\gamma \xrightarrow{F(\alpha_1)}{}' \gamma_1 \xrightarrow{F(\alpha_2)}{}' \ldots$ in $(\Gamma, T, \mathbb{A}', \longrightarrow')$.

The proof of the other direction is analogous, applying G instead of F. $\square$

The preservation of computations requires that the side-conditions of rules are unaffected by the label transformation. In practice, disciplined use of the general functions $set(\alpha, i, u)$, $get(\alpha, i)$ in side-conditions of rules, as illustrated in the next section, ensures this property. (The definition of a restricted metalanguage that would enforce such a discipline is left to future work.)

Corollary 1. *Let $\mathbb{A}$ be a label category constructed by applications of label transformers* $\boldsymbol{LabTrans}(j, \mathbb{B}_j)$ *for arbitrary (non-empty) categories* $\mathbb{B}_j$ *with* $j \in J \subset$ INDEX*. Let* Γ *and* T *be given, and let* $\longrightarrow$*) be specified by a set of rules* R *with labels ranging over* $\mathbb{A}$ *such that the index arguments of all applications of get and set in* R *are restricted to* J*. Let* $\mathbb{A}'$ *be the result of applying* $\boldsymbol{LabTrans}(i, \mathbb{B}_i)$ *to* $\mathbb{A}$*, where* $i \notin J$*, and let* $\longrightarrow'$ *be specified by* R *with labels ranging over* $\mathbb{A}'$*.*

Then for each computation $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \ldots$ *in* $(\Gamma, T, \mathbb{A}, \longrightarrow)$ *there is a corresponding computation* $\gamma \xrightarrow{\alpha_1'}{}' \gamma_1 \xrightarrow{\alpha_2'}{}' \ldots$ *in* $(\Gamma, T, \mathbb{A}', \longrightarrow')$*, and vice versa.*

Proof. The label transformer from $\mathbb{A}$ to $\mathbb{A} \times \mathbb{B}_i$ provides a projection functor $G : \mathbb{A} \times \mathbb{B}_i \to \mathbb{A}$, together with an embedding functor $F : \mathbb{A} \to \mathbb{A} \times \mathbb{B}_i$ for each object of $\mathbb{B}_i$. Both F and G preserve the values of terms of the form $get(\alpha, j)$ when $j \in J$, and commute with the operations $set(\alpha, j, u)$ and $\alpha_1 \mathbin{;} \alpha_2$, hence they preserve the holding of all side-conditions in R. The desired result follows. $\square$

6 Illustrative Examples

The fragments below are taken from a complete Modular SOS of ML concurrency primitives [28]. As in [4], we describe first a purely functional fragment, and extend it both with references and with processes. In the original SOS, each extension involved a complete reformulation of the rules given for the functional fragment; with MSOS, no such reformulation is needed, and the extensions below may be made in any order. For explanation of various details, see [28]

6.1 The Functional Fragment (excerpts)

Abstract Syntax

x	$\in$ VAR	variables
c	$\in$ CONST = BCONST $\cup$ FCONST	constants
b	$\in$ BCONST = ...	base constants
f	$\in$ FCONST = ...	function constants
e	$\in$ EXP	expressions
v	$\in$ VAL	values

$$
\begin{array}{lll}
e ::= & v & \text{value} \\
 & \mid\ x & \text{variable} \\
 & \mid\ e_1\ e_2 & \text{application} \\
 & \mid\ (e_1 \,.\, e_2) & \text{pair} \\
v ::= & c & \text{constant} \\
 & \mid\ (v_1 \,.\, v_2) & \text{pair value} \\
 & \mid\ \lambda x(e) & \lambda\text{-abstraction}
\end{array}
$$

Configurations

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

Label Transformers No label transformers are required here since we follow [4, 33, 34] and use syntactic substitution $e[x \mapsto v]$ instead of environments and closures.

Transition Rules

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

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

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{(e_1\,.\,e_2) \xrightarrow{\alpha} (e_1'\,.\,e_2)} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{(v_1\,.\,e_2) \xrightarrow{\alpha} (v_1\,.\,e_2')} \tag{3}$$

6.2 An Imperative Extension (excerpts)

The following extension of Sect. 6.1 provides ML-style references.

Abstract Syntax

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

Configurations

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

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

Label Transformers

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

where:

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

For stores, the notation $s[l \mapsto v]$ denotes the store that maps l to v, and otherwise maps locations l' to their values $s(l')$ according to s.

Transition Rules

$$\frac{s = \mathit{get}_{\mathit{pre}}(\iota, \mathit{store}) \quad l \in \mathit{dom}(s) \quad \alpha = \mathit{set}_{\mathit{post}}(\iota, \mathit{store}, s[l \mapsto v])}{\texttt{assign}\ (l\,.\,v) \xrightarrow{\alpha} \texttt{()}} \tag{4}$$

$$\frac{s = \mathit{get}_{\mathit{pre}}(\iota, \mathit{store}) \quad v = s(l)}{\texttt{deref}\ l \xrightarrow{\iota} v} \tag{5}$$

6.3 Concurrent Processes (excerpts)

The following extension of Sect. 6.1 (or of Sect. 6.2) provides CML-style process spawning and synchronization.

Abstract Syntax

$f \in \textsc{FConst} = \{\ldots, \texttt{receive}, \texttt{transmit}\}$	function constants
$p \in \textsc{Procs}$	processes
$e ::= \ldots$	
$\mid\ \texttt{sync}\ e$	synchronization
$\mid\ \texttt{spawn}\ e$	process creation
$p ::= e$	single process

Configurations

$\gamma ::= e \mid p$	arbitrary
$\tau ::= v$	terminal
$k \in \textsc{Chan}$	channel names
$ev \in \textsc{Event}$	event values
$v ::= \ldots$	
$\mid\ k$	channel name
$\mid\ ev$	event value
$ev ::= \ldots$	
$\mid\ k!v$	channel output
$\mid\ k?$	channel input
$p ::= \ldots$	
$\mid\ p_1 \parallel p_2$	concurrent process

Label Transformers

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

where:

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

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

Transition Rules

Expressions

$$\texttt{receive}\ k \xrightarrow{\iota} k? \qquad \texttt{transmit (}k\texttt{.}v\texttt{)} \xrightarrow{\iota} k!v \tag{6}$$

$$\frac{\alpha = set(\iota, acts, (ev, e))}{\texttt{sync}\ ev \xrightarrow{\alpha} e} \qquad \frac{\alpha = set(\iota, acts, v)}{\texttt{spawn}\ v \xrightarrow{\alpha} \texttt{()}} \tag{7}$$

Processes

$$\frac{p_1 \xrightarrow{\alpha} p_1'}{p_1 \parallel p_2 \xrightarrow{\alpha} p_1' \parallel p_2} \qquad \frac{p_2 \xrightarrow{\alpha} p_2'}{p_1 \parallel p_2 \xrightarrow{\alpha} p_1 \parallel p_2'} \tag{8}$$

$$\frac{\begin{array}{c} p_1 \xrightarrow{\alpha_1} p_1' \qquad p_2 \xrightarrow{\alpha_2} p_2' \\ \alpha_1 = set(\iota, acts, (ev_1, e_1)) \qquad \alpha_2 = set(\iota, acts, (ev_2, e_2)) \\ ev_1 \overset{k}{\asymp} ev_2 \text{ with } (e_1, e_2) \end{array}}{p_1 \parallel p_2 \xrightarrow{\iota} p_1' \parallel p_2'} \tag{9}$$

The relation $ev_1 \overset{k}{\asymp} ev_2$ with (e_1, e_2) holds when the events ev_1 and ev_2 match (on channel k) with respective results e_1 and e_2, as defined in [33, 34]. For instance, $k\texttt{!}v \overset{k}{\asymp} k\texttt{?}$ with $(\texttt{()}, v)$ holds.

$$\frac{e \xrightarrow{\alpha} e' \qquad \alpha = set(\iota, acts, v)}{e \xrightarrow{\iota} e' \parallel (v\ \texttt{()})} \tag{10}$$

7 Relation to Other Work

The modular approach to SOS presented here was inspired by the practical realization of Moggi's monad transformers [19] by Liang and Hudak in their modular monadic semantics framework [14], and by Wansbrough and Hamer's recent use [37] of that framework to give a modular monadic semantics of much of Action Notation, the original SOS definition of which [20] lacks modularity. Modular SOS 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 Modular SOS is expressed by the patterns of transitions in the rules (as in conventional SOS) and is not affected by label transformers. Thus there are no label transformers corresponding to exceptions or continuations.

This paper develops ideas first explored by the author in [22]. The technique of incorporating all semantic information in labels has previously been proposed as a general principle for SOS also by Degano and Priami [6], 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 the present approach.

There has been extensive work on various formats of small-step SOS (see a recent paper by Fokkink and Verhoef [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 [3], but modularity was not considered. The recent work of Turi and Plotkin [36] using coalgebraic techniques in SOS addresses foundational issues, and appears not to improve the modularity of semantic descriptions; moreover, the approach does not yet seem to be applicable to the description of conventional programming languages.

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. [7, 38]. The use of evaluation contexts appears to provide some inherent modularity, but obtaining full modularity may involve the introduction of many artificial internal steps [5]. Reppy's evaluation-context semantics for ML concurrency primitives [33, 34] has better modularity than the SOS given in [4]—see [28] for a detailed comparison of it with an MSOS for the same language. See also [25] for a more general survey of frameworks for logical specification of operational semantics.

8 Conclusion

The issue of modularity is significant for practical application of formal semantics. The structural approach to operational semantics is particularly popular for describing both conventional programming languages and process algebras, and it is widely taught to undergraduates [12, 29, 35]. Its poor modularity was left as an open problem by Plotkin [30, p.64]. The approach proposed in the present paper provides modularity in SOS through the use of a more disciplined meta-notation, while retaining the full generality of Plotkin's original framework. The fundamental label transformers of MSOS incorporate the standard techniques used in SOS, in much the same way as monad transformers in denotational semantics incorporate standard techniques for constructing domains. All this is obtained through a simple (yet apparently novel) combination of the familiar notions of labelled transition system and category.

A higher-level approach to obtaining modularity in operational semantics, called Action Semantics, has previously been proposed by the author, in collaboration with Watt [1, 20, 21]. It employs a rich semantic notation called Action Notation, whose operational semantics was originally defined using SOS [20, Apps. B–C]. The lack of modularity of that SOS has hindered the definition of extensions or variants of Action Notation. An MSOS of Action Notation has recently been developed [26], and its modularity is greatly facilitating the reconsideration of the detailed design of Action Notation [21, Sect. 8].

The full MSOS descriptions of Action Notation [26] and of ML concurrency primitives [28] should provide sufficient evidence of the benefits of MSOS as a descriptive framework, and of the way that it scales up smoothly to richer languages. Some points need further investigation:

- It is claimed that MSOS can be applied just as well to big-step as to small-step operational semantics. It would be interesting to test this claim by reformulating the definition of Standard ML [18] using MSOS.
- Label categories may be equipped with further operations for composing labels, such as parallel composition. It seems possible to define a label category appropriate for a big-step modelling of interleaving (the labels being sets of sequences of disconnected small steps, with interleaving corresponding to nondeterministic shuffling of sequences, and indivisibility corresponding to selecting just those sequences where the steps are connected). However, it is unclear whether the simple notion of label transformer presented here can be generalized to construct such label categories.
- The precise relationship between MSOS and Enhanced Operational Semantics [6, 32] is unclear, especially regarding how best to deal with proof terms in the former, and with the description of imperative features in the latter.
- It appears that bisimulations can be lifted along label transformers. Thus after applying a label transformer, previously-proved semantic equivalences based on bisimulation should remain valid.
- It should be checked whether the bisimulation theory obtained using the MSOS for ML concurrency primitives [28] is comparable to that obtained for the same language using conventional SOS [8].

Acknowledgements Thanks to the anonymous reviewers for suggestions of improvements, and to Søren B. Lassen and Carolyn Talcott for comments on an earlier version. Thanks also to José Meseguer for helpful suggestions concerning the presentation of label transformers.

During the work reported here, the author was supported by BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark; by an International Fellowship from SRI International; and by DARPA-ITO through NASA-Ames contract NAS2-98073.

References

1. Action semantics. Home page: `http://www.brics.dk/Projects/AS/`.
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-Verlag, 1991.

3. K. L. Bernstein. A congruence theorem for structured operational semantics of higher-order languages. In *Proc. LICS'98*, pages 153–163. IEEE, 1998.
4. 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.
5. 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*, volume 789 of *LNCS*, pages 244–272, Sendai, Japan, 1994. Springer-Verlag.
6. P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, June 1996.
7. M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 193–217. North-Holland, 1987.
8. W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisumulation for core CML. *J. Functional Programming*, 8(5):447–451, 1998.
9. W. Fokkink and R. van Glabbeek. Ntyft/ntyxt rules reduce to ntree rules. *Information and Computation*, 126(1):1–10, 1996.
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 *Proof, Language, and Interaction*. The MIT Press, 1999. To appear.
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*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.
14. S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *LNCS*, pages 219–234. Springer-Verlag, 1996.
15. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
16. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
17. R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
19. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.
20. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
21. P. D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, 1996.
22. P. D. Mosses. Semantics, modularity, and rewriting logic. In *WRLA'98, Proc. 2nd Int. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, France*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

23. P. D. Mosses. Foundations of modular SOS. Research Series BRICS-RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/54`. Full version of [24].
24. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available [23].
25. P. D. Mosses. Logical specification of operational semantics. In *CSL'99, Proc. Conf. on Computer Science Logic*, volume 1683 of *LNCS*, pages 32–49. Springer-Verlag, 1999. Also available at `http://www.brics.dk/RS/99/55`.
26. P. D. Mosses. A modular SOS for Action Notation. Research Series BRICS-RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/56`. Full version of [27].
27. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In P. D. Mosses and D. A. Watt, editors, *AS'99, Proc. Second International Workshop on Action Semantics, Amsterdam, The Netherlands*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, May 1999. Full version available [26].
28. P. D. Mosses. A modular SOS for ML concurrency primitives. Research Series BRICS-RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/57`.
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. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
31. G. D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*. IFIP, North-Holland, 1983.
32. C. Priami. *Enhanced Operational Semantics for Concurrency*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Pisa, 1996.
33. 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.
34. J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Computer Science Dept., Cornell Univ., 1992. Tech. Rep. TR 92-1285.
35. K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.
36. D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *Proc. LICS'97*. IEEE, 1997.
37. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997.
38. A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Dept. of Computer Science, Rice University, 1991.

Recent BRICS Report Series Publications

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

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

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

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

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

RS-99-49 **Alexander A. Ageev and Maxim I. Sviridenko.** ***An Approximation Algorithm for Hypergraph Max k-Cut with Given Sizes of Parts.*** **December 1999. 12 pp.**

RS-99-48 **Rasmus Pagh.** ***Faster Deterministic Dictionaries.*** **December 1999. 14 pp. To appear in** ***The Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*****, SODA '00 Proceedings, 2000.**

RS-99-47 **Peter Bro Miltersen and Vinodchandran N. Variyam.** ***Derandomizing Arthur-Merlin Games using Hitting Sets.*** **December 1999. 21 pp. Appears in Beame, editor,** ***40th Annual Symposium on Foundations of Computer Science*****, FOCS '99 Proceedings, 1999, pages 71–80.**