BRICS

Basic Research in Computer Science

BRICS RS-99-55 P. D. Mosses: Logical Specification of Operational Semantics

Logical Specification of Operational Semantics

Peter D. Mosses

BRICS Report Series **RS-99-55**

ISSN 0909-0878 **December 1999**

Copyright © 1999, Peter D. Mosses.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

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/99/55/`

Logical Specification of Operational Semantics ⋆

Peter D. Mosses

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

Abstract. Various logic-based frameworks have been proposed for specifying the operational semantics of programming languages and concurrent systems, including inference systems in the styles advocated by Plotkin and by Kahn, Horn logic, equational specifications, reduction systems for evaluation contexts, rewriting logic, and tile logic.
We consider the relationship between these frameworks, and assess their respective merits and drawbacks—especially with regard to the modularity of specifications, which is a crucial feature for scaling up to practical applications. We also report on recent work towards the use of the Maude system (which provides an efficient implementation of rewriting logic) as a meta-tool for operational semantics.

1 Introduction

The designers, implementors, and users of a programming language all need to acquire an intrinsically *operational* understanding of its semantics. Programming language reference manuals attempt to provide such an understanding using informal, natural language; but they are prone to ambiguity, inconsistency, and incompleteness, and totally unsuitable as a basis for sound reasoning about the effects of executing programs—especially when concurrency is involved.

Various mathematical frameworks have been proposed for giving formal descriptions of programming language semantics. Denotational semantics generally tries to avoid direct reference to operational notions, and its abstract domain-theoretic basis remains somewhat inaccessible to most programmers (although modelling programs as higher-order functions has certainly given useful insight to language designers and to theoreticians). Operational semantics, which directly aims to model the program execution process, is generally based on familiar first-order notions; it has become quite popular, and has been preferred to denotational semantics for defining programming languages [28] and process algebras [26].

Despite the relative popularity of operational semantics, there have been some "semantic engineering" problems with scaling up to descriptions of full practical programming languages. A significant feature that facilitates scaling-up

⋆ Written while visiting SRI International and Stanford University. Published in *CSL 99, Computer Science Logic*, volume 1683 of *LNCS*, pages 32–49. Springer-Verlag, 1999.

is good modularity: the formulation of the description of one construct should not depend on the presence (or absence) of other constructs in the language. Recently, the author has proposed a solution to the modularity problem for the structural approach to operational semantics [31, 33].

There are different ways of specifying operational semantics for a programming language: an *interpreter* for programs—written in some (other) programming language, or defined mathematically as an abstract machine—is an *algorithmic* specification, determining *how* to execute programs; a *logic* for inferring judgements about program executions is a *declarative* specification, determining *what* program executions are allowed, but leaving how to find them to logical inference. Following Plotkin's seminal work [38], much interest has focussed on logical specification of operational semantics.

In fact various kinds of logic have been found useful for specifying operational semantics: arbitrary inference systems, natural deduction systems, Horn logic, equational logic, rewriting logic, and tile logic, among others. Sections 2 and 3 review and consider the relationship between these applied logics, pointing out some of their merits and drawbacks—especially with regard to the modularity of specifications. The brief descriptions of the various logics are supplemented by illustrative examples of their use. It is hoped that the survey thus provided will be useful as an introduction to the main techniques available for logical specification of operational semantics.

The inference of a program execution in some logic is clearly not the same thing as the inferred execution itself. Nevertheless, a system implementing logical inference may be used to execute programs according to their operational semantics. Section 4 reports on recent work towards the use of the Maude system (which provides an efficient implementation of rewriting logic) as a meta-tool for operational semantics.

2 Varieties of Structural Operational Semantics

The structural style of operational semantics (SOS) is to specify inference rules for *steps* (or transitions) that may be made not only by whole programs but also by their constituent phrases: expressions, statements, declarations, etc. The steps allowed for a compound phrase are generally determined by the steps allowed for its component phrases, i.e., the steps are defined inductively according the (abstract) syntax of the described programming language. An atomic assertion of the specified logic (such as $\gamma \longrightarrow \gamma'$) asserts the possibility of a step from one *configuration* γ to another γ'. Some configurations are usually distinguished as *terminal*, and have no further steps, whereas initial and intermediate configurations have phrases that remain to be executed as components.

Small-step SOS: In so-called small-step SOS [38], a single step for an atomic phrase often gives rise to a single step for its enclosing phrase (and thus ultimately for the whole program). A complete program execution is modelled as a succession—possibly infinite—of these small steps. During such a program execution, phrases whose execution has terminated get replaced by the values that they have computed.

Suppose that an abstract syntax for expressions e includes also values v:

$$\begin{aligned} e &::= v \mid \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \mid \ldots \\ v &::= \mathit{true} \mid \mathit{false} \mid \ldots \end{aligned}$$

Here are a couple of typical examples of inference rules for small-step SOS, to illustrate the above points:

$$\frac{e_0 \longrightarrow e_0'}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \longrightarrow \texttt{if } e_0' \texttt{ then } e_1 \texttt{ else } e_2} \tag{1}$$

$$\texttt{if } \mathit{true} \texttt{ then } e_1 \texttt{ else } e_2 \longrightarrow e_1 \qquad \texttt{if } \mathit{false} \texttt{ then } e_1 \texttt{ else } e_2 \longrightarrow e_2 \tag{2}$$

In the lack of further rules for $\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2$, it is easy to see that the intended operational semantics has been specified: the sub-expression e_0 must be executed first, and if that execution terminates with a truth-value, only one of e_1, e_2 will then be executed.

Big-step SOS: In big-step SOS [17], a step for a phrase always corresponds to its entire (terminating) execution, so no iteration of steps is needed. A step for a compound phrase thus depends on steps for all those component phrases that have to be executed. (Big-step SOS has been dubbed *Natural Semantics* since the inference rules may resemble those of Natural Deduction proof systems [17].) Here is an example, where the notation $e \Downarrow v$ asserts the possibility of the evaluation (i.e., execution) of e terminating with value v:

$$\frac{e_0 \Downarrow \mathit{true} \qquad e_1 \Downarrow v_1}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v_1} \qquad \frac{e_0 \Downarrow \mathit{false} \qquad e_2 \Downarrow v_2}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v_2} \tag{3}$$

The intended operational semantics, where e_0 is supposed to be executed before e_1 or e_2, is not so evident here as it is in the small-step SOS rules. In other examples, however, explicit data dependencies may indicate the flow of control more clearly.

Big-step SOS cannot express the possibility of non-terminating executions, and thus it appears ill-suited to the description of reactive systems. However, the possibility of non-termination may be specified separately [8].

Note that small- and big-step SOS may be used together in the same description, e.g. big-step for modelling expression evaluation and small-step for modelling statement execution. Moreover, the transitive closure of the small-step relation (restricted to appropriate types of arguments) provides the big-step relation between phrases and their computed values.

Substitution: Binding constructs of programming languages, such as declarations and formal parameters, give rise to open phrases with free variables; however, these phrases do not get executed until the values to be bound to the free variables have actually been determined. Thus one possibility is to replace the free variables by their values, producing a closed phrase, using a *substitution* operation (here written $[v/x]e$). However, the definition of substitution itself can be somewhat tedious—in practice, it is often left to the reader's imagination (as here):

$$\frac{e_1 \longrightarrow e_1'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2} \tag{4}$$

$$\texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow [v_1/x]e_2 \tag{5}$$

There is obviously no need to give a rule for evaluating a variable x to its value when using substitution.

Environments: An alternative approach, inspired by the treatment of binding constructs in denotational semantics and in Landin's work [18], is to use *environments* ρ: a judgement then has the form $\rho \vdash \gamma \longrightarrow \gamma'$. In effect, the environment keeps track of the relevant substitutions that could have been made; the combination (often referred to as a *closure*) of an open phrase and an appropriate environment is obviously equivalent to a closed phrase. Environments are particularly simple to use in big-step SOS, but in small-step SOS, auxiliary syntax for explicit closures may have to be added to the described language (Plotkin managed to avoid adding auxiliary syntax in [38] only because the example language that he described already had a form of local declaration that was general enough to express closures). Here is the same example as described above, but now using environments instead of substitution:

$$\frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2} \tag{6}$$

$$\frac{\rho[x \mapsto v_1] \vdash e_2 \longrightarrow e_2'}{\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = v_1\ \texttt{in}\ e_2'} \tag{7}$$

$$\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ v_2 \longrightarrow v_2 \tag{8}$$

Here, in contrast to when using substitution, a rule is needed for evaluating the use of a variable x occurring in an expression e:

$$\frac{\rho(x) = v}{\rho \vdash x \longrightarrow v} \tag{9}$$

The equation $\rho(x) = v$ above is formally regarded as a *side-condition* on the inference rule, although for notational convenience it is written as an antecedent of the rule. It restricts the conclusion of the rule to the case that the environment ρ does indeed provide a value v for x. Note that proofs of steps do not explicitly involve proofs of side-conditions.

Stores: For describing imperative programming languages, where the values last assigned to variables have to be kept for future reference, configurations are usually pairs of phrases and *stores*. Thus a judgement might have the form $\rho \vdash e, s \longrightarrow e', s'$. Stores themselves are simply (finite) maps from locations to stored values. Unfortunately, adding stores to configurations invalidates all our previous rules, which should now be reformulated before being extended with rules for imperative phrases. For instance:

$$\frac{\rho \vdash e_1, s \longrightarrow e_1', s'}{\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2, s \longrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2, s'} \tag{10}$$

$$\frac{\rho[x \mapsto v_1] \vdash e_2, s \longrightarrow e_2', s'}{\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2, s \longrightarrow \texttt{let}\ x = v_1\ \texttt{in}\ e_2', s'} \tag{11}$$

$$\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ v_2, s \longrightarrow v_2, s \tag{12}$$

$$\frac{\rho(x) = v}{\rho \vdash x, s \longrightarrow v, s} \tag{13}$$

The need for this kind of reformulation reflects the poor inherent modularity of SOS. Later in this section, however, we shall see how the modularity of SOS can be significantly improved.

The following rules illustrate the SOS description of variable allocation, assignment, and dereferencing (assuming that locations l are not values v):

$$\frac{l \notin \mathrm{dom}(s)}{\rho \vdash \texttt{ref}\ v, s \longrightarrow l, s[l \mapsto v]} \tag{14}$$

$$\frac{l \in \mathrm{dom}(s)}{\rho \vdash l\ \texttt{:=}\ v, s \longrightarrow (\ ), s[l \mapsto v]} \tag{15}$$

$$\frac{\rho(x) = l \qquad s(l) = v}{\rho \vdash x, s \longrightarrow v, s} \tag{16}$$

Conventions: A major example of an operational semantics of a programming language is the definition of Standard ML (SML) [28]. It is a big-step SOS, using environments and stores. A couple of "conventions" have been introduced to abbreviate the rules: one of them allows the store to be elided from configurations, relying on the flow of control to sequence assignments to variables; the other caters for raised exceptions preempting the normal sequence of evaluation of expressions. Although these conventions achieve a reasonable degree of conciseness, the need for them perhaps indicates that the big-step style of SOS has some pragmatic problems with scaling up to languages such as SML. Moreover, they make it difficult to exploit the definition of SML directly for verification or prototyping.

Recently, an alternative definition of SML has been proposed [15], without the need for the kind of conventions used in the original definition. SML is first translated into an “internal language”, which is itself defined by a (small-step) reduction semantics, see Sect. 3. (The translation of SML to the internal language is itself specified using a big-step SOS, but that aspect of the approach seems to be inessential.) A similar technique is used in the action semantics framework [29, 30], where programs are mapped to an action notation that has already been defined using a (small-step) SOS.

Process Calculi: Small-step SOS is a particularly popular framework for the semantic description of calculi for concurrent processes, such as CCS. There, steps are generally *labelled*, and judgements have the form $\gamma \xrightarrow{\alpha} \gamma'$. For CCS, labels α range over atomic “actions”, and for each action l there is a complementary action $\bar{l}$ for synchronization; there is also an unobservable label τ, representing an internal synchronization. Here are some of the usual rules for CCS [26]:

$$\alpha.p \xrightarrow{\alpha} p \tag{17}$$

$$\frac{p_1 \xrightarrow{\alpha} p_1'}{p_1 \mid p_2 \xrightarrow{\alpha} p_1' \mid p_2} \qquad \frac{p_2 \xrightarrow{\alpha} p_2'}{p_1 \mid p_2 \xrightarrow{\alpha} p_1 \mid p_2'} \tag{18}$$

$$\frac{p_1 \xrightarrow{l} p_1' \qquad p_2 \xrightarrow{\bar{l}} p_2'}{p_1 \mid p_2 \xrightarrow{\tau} p_1' \mid p_2'} \tag{19}$$

Also programming languages with constructs for concurrency, for instance Concurrent ML [41, 40], can be described using small-step SOS. Unfortunately, the SOS description proposed for ML with concurrency primitives in [2] is not inductive in the syntax of the language, and the need to reformulate inference rules previously given for the purely functional part of the language is again a sign of the poor inherent modularity of the SOS framework. Also the more conventional SOS descriptions given in [12, 16] have undesirably complex rules for the functional constructs.

Syntactic Congruence: When using SOS to describe process calculi, it is common practice to exploit a *syntactic congruence* on phrases, i.e., the syntax becomes a set of equivalence classes. For instance, the processes $p_1 \mid p_2$ and $p_2 \mid p_1$ might be identified, removing the need for one of the symmetric rules given in (18) above.

Evaluation to Committed Form: It is possible to describe the operational semantics of CCS and other concurrent calculi without labelling steps [36]. The idea is to give a big-step SOS for the evaluation of a process to its “committed forms” where the possible actions are apparent (cf. reduction to “head normal form” in the lambda-calculus). For example:

$$\frac{p_1 \Downarrow l.p_1' \qquad p_2 \Downarrow \bar{l}.p_2' \qquad p_1' \mid p_2' \Downarrow k}{p_1 \mid p_2 \Downarrow k} \tag{20}$$

The technique relies heavily on a syntactic congruence between processes.

Enhanced Operational Semantics: By labelling steps with their *proofs*, information about features such as causality and locality can be provided. This idea has been further developed and applied in the "enhanced" SOS style [9, 39], where models taking account of different features of concurrent systems can be obtained by applying relabelling functions (extracting the relevant details from the proofs).

For a simple CCS-like process calculus, proofs may be constructed using tags $|_1$, $|_2$ to record the use of the rules that let processes act alone, and pairs $\langle |_1\,\theta_1, |_2\,\theta_2 \rangle$ to record synchronization. An auxiliary function l is used to extract actions from proofs. The following rules illustrate the form of judgements:

$$\frac{p_1 \xrightarrow{\theta} p_1'}{p_1 \mid p_2 \xrightarrow{|_1\theta} p_1' \mid p_2} \qquad \frac{p_2 \xrightarrow{\theta} p_2'}{p_1 \mid p_2 \xrightarrow{|_2\theta} p_1 \mid p_2'} \tag{21}$$

$$\frac{p_1 \xrightarrow{\theta_1} p_1' \qquad p_2 \xrightarrow{\theta_2} p_2' \qquad l(\theta_1) = \overline{l(\theta_2)}}{p_1 \mid p_2 \xrightarrow{\langle |_1\theta_1, |_2\theta_2 \rangle} p_1' \mid p_2'} \tag{22}$$

Despite its somewhat intricate notation, enhanced operational semantics provides a welcome uniformity and modularity for models of concurrent systems. By using substitution, the need for explicit environments can be avoided—but if one wanted to add stores, it seems that a major reformulation of the inference rules for steps would still be required. Or could labels be used also to record *changes* to stored values? The next variety of SOS considered here suggests that they can indeed.

Modular SOS: Recently, the author has proposed a solution to the SOS modularity problem [31, 33]. In Modular SOS (MSOS) the transition rules for each construct are completely independent of the presence or absence of other constructs. 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.

The basic idea of MSOS is to incorporate *all* semantic entities as components of labels. Thus configurations are restricted to syntax and computed values, and judgements are *always* of the form $\gamma \xrightarrow{\alpha} \gamma'$.

In fact the labels in MSOS are regarded as the arrows of a *category*, and the labels on adjacent steps have to be composable in that category. The labels are no longer the simple atomic actions often used in studies of process algebra, but usually have semantic entities—e.g. environments and stores—as components; so do the objects of the label category, which correspond to the states of the processed information.

Some basic label transformers for defining appropriate categories (starting from the trivial category) are available; they correspond to some of the simpler monad transformers used to obtain modularity in denotational semantics. Each label transformer adds a fresh indexed component to labels, and provides notation for setting and getting that component—independently of the presence or absence of other components. By using variables α ranging over arbitrary labels, and ι ranging over arbitrary *identity* labels that remain in the same state, rules can be expressed independently of the presence or absence of irrelevant components of labels. For example:

$$\frac{e_0 \xrightarrow{\alpha} e_0'}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\alpha} \texttt{if } e_0' \texttt{ then } e_1 \texttt{ else } e_2} \tag{23}$$

$$\texttt{if } \mathit{true} \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\iota} e_1 \qquad \texttt{if } \mathit{false} \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\iota} e_2 \tag{24}$$

The above rules remain both valid and appropriate when the category of labels gets enriched with (e.g.) environment components, allowing the rules for binding constructs to be added:

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{\texttt{let } x = e_1 \texttt{ in } e_2 \xrightarrow{\alpha} \texttt{let } x = e_1' \texttt{ in } e_2} \tag{25}$$

$$\frac{\rho = \mathit{get}(\alpha, \mathit{env}) \qquad \alpha' = \mathit{set}(\alpha, \mathit{env}, \rho[x \mapsto v_1]) \qquad e_2 \xrightarrow{\alpha'} e_2'}{\texttt{let } x = v_1 \texttt{ in } e_2 \xrightarrow{\alpha} \texttt{let } x = v_1 \texttt{ in } e_2'} \tag{26}$$

$$\texttt{let } x = v_1 \texttt{ in } v_2 \xrightarrow{\iota} v_2 \tag{27}$$

$$\frac{\rho = \mathit{get}(\iota, \mathit{env}) \qquad \rho(x) = v}{x \xrightarrow{\iota} v} \tag{28}$$

The use of ι rather than α above excludes the possibility of any change of state.

Axiomatic Specifications: For proof-theoretic reasoning about SOS descriptions—especially when establishing bisimulation and other forms of equivalence—it is convenient that steps can only occur when proved by just the *specified* inference rules. For other purposes, however, it may be an advantage to reformulate the inference rules of SOS as ordinary conditional formulae, i.e., *Horn clauses*, and use the familiar inference rules for deduction, such as *Modus Ponens*. The close correspondence between inference rules and Horn clauses has been used in the implementation of big-step SOS by compilation to Prolog [17].

The axiomatic reformulation of SOS requires side-conditions on rules to be treated as ordinary conditions, along with judgements about possible steps. It has been adopted in the SMoLCS framework [1], which combines SOS with algebraic specifications. It has also been exploited in the modular SOS of action notation [35], where CASL, the Common Algebraic Specification Language [7], is used throughout (CASL allows the declaration of total and partial functions, relations, and subsorts, which is just what is needed in the side-conditions of SOS descriptions).

3 Varieties of Reduction Semantics

Many of the inference rules specified in the structural approach to operational semantics merely express that execution steps of particular components give rise to execution steps of the enclosing phrases. The notion of *reduction* in term rewriting systems enjoys a similar property, except that there is *a priori* no restriction on the order in which component phrases are to be reduced. Thus for any term constructor f, the following inference rule may be specified for the reduction relation $t \longrightarrow t'$:

$$\frac{t_i \longrightarrow t'_i}{f(t_1, \ldots, t_i, \ldots, t_n) \longrightarrow f(t_1, \ldots, t'_i, \ldots, t_n)} \tag{29}$$

The above rule is subsumed by the following somewhat more elegant rule, where C ranges over arbitrary one-hole term *contexts*, and $C[t]$ is the term obtained by filling the unique hole in the context C with the term t:

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \tag{30}$$

It is straightforward to define the arbitrary one-hole contexts for any ranked alphabet of (constant and) function symbols; similarly for many-sorted and order-sorted signatures—introducing a different sort of context for each *pair* of argument and result sorts.

Several frameworks for operational semantics are based on variations of the basic notion of reduction, and are reviewed below.

Reduction Strategies: The problem with using ordinary reduction to specify operational semantics is the lack of control concerning the order of reduction steps: the entire sequence of reductions might be applied to a part of the program that in fact should not be executed at all.

For instance, consider the λ-expressions with constants, which may be regarded as a simple functional programming language:

$$\begin{aligned} e &::= v \mid e_1\ e_2 \\ v &::= b \mid f \mid x \mid \lambda x.e \end{aligned}$$

where the basic constants b and function constants f are left unspecified. The execution steps for evaluating λ-expressions are δ-reductions, concerned with applications of the form $f\ b$, and β-reductions:

$$(\lambda x.e)(e') \longrightarrow [e'/x]e \tag{31}$$

where the substitution of expressions e' for variables x, written $[e'/x]e$, is assumed to avoid capture of free variables. An expression such as

$$(\lambda y.b)((\lambda x.xx)(\lambda x.xx))$$

has both terminating and non-terminating reduction sequences: the one that takes the leftmost, outermost β-reduction corresponds to "call-by-name" semantics for λ-expressions; that which always applies β-reduction to $(\lambda x.xx)(\lambda x.xx)$ corresponds to "call-by-value" semantics.

Standard reduction sequences are those which always make the leftmost outermost reduction at each step. For λ-expressions, restricting reductions to standard δ- and β-reductions ensures call-by-name operational semantics. Remarkably, also call-by-value semantics can be ensured by restricting to standard reductions—provided that the β-reduction rule is itself restricted to the case where the argument of the application is already a value v [37, 11]:

$$(\lambda x.e)(v) \longrightarrow [v/x]e \tag{32}$$

Standard reduction sequences with this restricted notion of β-reduction correspond to an operational semantics for λ-expressions defined by an SECD machine [37].

By adopting the restriction to standard reductions, it might be possible to give reduction semantics for other programming languages. However, the following technique not only subsumes this approach, but also has the advantage of admitting an explanation in terms of inference systems.

Evaluation Contexts: An alternative way of controlling the applicability of reductions is to require them to occur in *evaluation contexts* [10]. It is convenient to specify evaluation contexts E in the same way as the abstract syntax of programs, using context-free grammars. The symbol [ ] represents the hole of the context; the grammar must ensure that exactly one hole occurs in any evaluation context.

The restriction to evaluation contexts corresponds to simply replacing the general context rule for reduction (30) above by:

$$\frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']} \tag{33}$$

For example, to obtain the call-by-value semantics of λ-expressions, let evaluation contexts E be defined by the grammar:

$$E ::= [\;] \mid v\ E \mid E\ e$$

It is easy to see that when an expression is of the form $E[e_1\ e_2]$, a standard reduction step can only reduce $e_1\ e_2$ or some sub-expression of it. Similarly, it appears that call-by-name semantics can be obtained by reverting to ordinary β-reduction (31) and letting evaluation contexts be defined as follows:

$$E ::= [\;] \mid f\ E \mid E\ e$$

(where f ranges over function constants).

The following specification of evaluation contexts would be appropriate for the intended operational semantics of the illustrative language constructs considered in Sect. 2:

$$E ::= [\,] \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = E\ \texttt{in}\ e_2$$

(where the grammar for expressions e and values v is as before). Notice the close correspondence between the productions of the above grammar and the previously-given small-step SOS rules (1) and (4). The above grammar is clearly more concise than the inference rules for expressing the allowed order of execution; this economy of specification may account for at least some of the popularity of the evaluation context approach.

Assuming that reductions are restricted to occur only in evaluation contexts, the following rules may now be given:

$$\texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1 \qquad\qquad \texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2 \tag{34}$$

$$\texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow [v_1/x]e_2 \tag{35}$$

where $[v/x]e$ is substitution, as before. As the reader may have noticed, these are exactly the same as the small-step SOS rules (2) and (5).

An alternative technique with evaluation contexts is to combine (33) above with the reduction rules themselves—now insisting that reductions are always applied to the entire program. With the same definition of evaluation contexts, the above reduction rules would then be written:

$$E[\texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \longrightarrow E[e_1] \tag{36}$$

$$E[\texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \longrightarrow E[e_2] \tag{37}$$

$$E[\texttt{let}\ x = v_1\ \texttt{in}\ e_2] \longrightarrow E[[v_1/x]e_2] \tag{38}$$

This seemingly innocent reformulation in fact provides a significant new possibility, which is perhaps the *forte* of the evaluation context approach: reductions may depend on and/or change the structure of the context itself. For example, we may easily add a construct that is intended to stop the execution of the entire program, and specify the reduction:

$$E[\texttt{stop}] \longrightarrow \texttt{stop} \tag{39}$$

To specify the same semantics in small-step semantics would require giving an explicit basic rule for the propagation of `stop` out of each evaluation context construct:

$$\texttt{if}\ \texttt{stop}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow \texttt{stop} \tag{40}$$

$$\texttt{let}\ x = \texttt{stop}\ \texttt{in}\ e_2 \longrightarrow \texttt{stop} \tag{41}$$

In a big-step semantics, one would have to provide extra inference rules, e.g.:

$$\frac{e_0 \Downarrow \texttt{stop}}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow \texttt{stop}} \tag{42}$$

Moreover, evaluation contexts can be used to specify the operational semantics of advanced control constructs, such as those manipulating *continuations* [11]. Although it may be possible to specify continuations in SOS, the appropriateness of the use of evaluation contexts here cannot be denied.

An evaluation context may contain more than just the syntactic control context: for instance, it may also contain a store, recording the values assigned to variables. A store s is represented syntactically as a sequence of pairs of locations l and values v, with no location occurring more than once. It is quite straightforward to give reduction rules for variable allocation, assignment, and dereferencing [4]:

$$s\ E[\texttt{ref}\ v] \longrightarrow s,(l,v)\ E[l] \quad \text{if } l \text{ is not used in } s \tag{43}$$

$$s,(l,v),s'\ E[l\ \texttt{:=}\ v'] \longrightarrow s,(l,v'),s'\ E[(\ )] \tag{44}$$

$$s,(l,v),s'\ E[l] \longrightarrow s,(l,v),s'\ E[v] \tag{45}$$

The previously given rules for functional constructs using explicit evaluation contexts (e.g. (36)) remain valid, so the modularity of the approach appears to be good—also when adding concurrency primitives to a functional language, as illustrated in [40, 41]. However, it appears that it would not be so straightforward to add explicit environments to evaluation contexts, and the reliance on syntactic substitution may complicate the description of languages with "dynamic" scope rules.

One significant potential problem when using evaluation contexts for modelling the operational semantics of concurrent languages is how to define and prove equivalence of processes. In particular cases "barbed" bisimulation can be defined [27, 42]; also, a rather general technique for extracting labelled transition systems (and hence bisimulations) from evaluation context semantics has been proposed [43].

Rewriting Logic: The framework of Rewriting Logic (RL) [21] generalizes conventional term rewriting in two main directions:

- rewriting may be *modulo* a set of equations between terms (i.e., it applies to arbitrary equationally-specified data structures); and
- rewriting may be *concurrent* (i.e., non-overlapping sub-terms may be rewritten simultaneously).

Moreover, no assumptions about confluence or termination are made: the rules are understood not as equations, but as transitions.

The inference rules for RL are as follows, where rewriting from between equivalence classes of terms is written $[t] \longrightarrow [t']$. Rewriting is taken to be reflexive:

$$[t] \longrightarrow [t] \tag{46}$$

which allows one or more of the arguments to remain the same in concurrent rewriting:

$$\frac{[t_1] \longrightarrow [t_1'] \quad \dots \quad [t_n] \longrightarrow [t_n']}{[f(t_1,\dots,t_n)] \longrightarrow [f(t_1',\dots,t_n')]} \tag{47}$$

The following inference rule combines replacement of variables $x_1, \ldots, x_m$ by terms $t_1, \ldots, t_m$ in a specified rule $r : [t(x_1, \ldots, x_m)] \longrightarrow [t'(x_1, \ldots, x_m)]$ with the possibility of rewriting the terms in the same step:

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[t(t_1/x_1, \ldots, t_n/x_n)] \longrightarrow [t'(t'_1/x_1, \ldots, t'_n/x_n)]} \tag{48}$$

Finally, rewriting is taken to be transitive:

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]} \tag{49}$$

Specified rewriting rules are also allowed to be conditional, which requires a further inference rule for discharging conditions.

RL has been used as a unifying model for concurrency [23] and as a logical framework [20]. It has also been proposed as a *semantic* framework, as an alternative to frameworks such as SOS [20]. RL has been efficiently implemented in the Maude system [5], which makes its use as a semantic framework particularly attractive in connection with the possibilities for prototyping semantic descriptions (see Sect. 4).

Two techniques for expressing SOS descriptions in RL have been proposed [20]. The first is a special case of a general technique for representing sequent systems in unconditional RL, with the rewriting relation corresponding to provability; the second is more specific to SOS, and uses conditional rewriting rules. Let us illustrate both techniques with the same example: the SOS rules (17)–(19) for concurrency in CCS (Sect. 2).

To start with, term constructors for the abstract syntax of processes and labels are needed; we shall only make use of the binary process constructor $p \mid p'$, which is now specified to be both associative and commutative (corresponding to a syntactic congruence in SOS).

For the first technique, we also introduce a term constructor $S(p, \alpha, p')$ representing the assertion of an SOS step from process p to process p' with label α; and an infix term constructor $s_1 \& s_2$ representing the conjunction of such assertions, specified to be associative, commutative, with unit $\perp$.

The SOS rules are then expressed in RL as follows:

$$[\perp] \longrightarrow [S(\alpha.p, \alpha, p)] \tag{50}$$

$$[S(p_1, \alpha, p'_1)] \longrightarrow [S(p_1 \mid p_2, \alpha, p'_1 \mid p_2)] \tag{51}$$

$$[S(p_1, l, p'_1) \& S(p_2, \bar{l}, p'_2)] \longrightarrow [S(p_1 \mid p_2, \tau, p'_1 \mid p'_2)] \tag{52}$$

The relationship between the SOS steps and the rewriting relation is that $p \xrightarrow{\alpha} p'$ in the SOS iff $[\perp] \longrightarrow [S(p, \alpha, p')]$ is provable in RL. Note that the rewriting relation is highly non-deterministic, and in practice a goal-directed strategy would be needed in order to use the Maude implementation of RL for proving $[\perp] \longrightarrow [S(p, \alpha, p')]$ for some particular process p.

For the second technique, we introduce only a term constructor $\alpha;p$ that combines the label α with the process p, representing the "result" of some SOS step. The result sort of $\alpha;p$ is regarded as a supersort of the sort of processes, and rewriting is always of the form $[p] \longrightarrow [\alpha;p']$, i.e. it is sort-increasing. The SOS rules are then expressed in RL as follows:

$$[\alpha.p] \longrightarrow [\alpha;p] \tag{53}$$

$$[p_1 \mid p_2] \longrightarrow [\alpha;p_1' \mid p_2] \;\text{ if }\; [p_1] \longrightarrow [\alpha;p_1'] \tag{54}$$

$$[p_1] \mid p_2] \longrightarrow [\tau;p_1' \mid p_2'] \;\text{ if }\; [p_1] \longrightarrow [l;p_1'] \wedge [p_2] \longrightarrow [\bar{l};p_2'] \tag{55}$$

The relationship between the SOS steps and the rewriting relation is now that given a process p, there are processes $p_1, \ldots, p_{n-1}$ and labels $\alpha_1, \ldots, \alpha_n$ such that $p \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_1} \ldots p_{n-1} \xrightarrow{\alpha_n} p'$ in SOS iff $[p] \longrightarrow [\alpha_1; \ldots; \alpha_n; p']$ in RL.

Tile Logic: Although Tile Logic (TL) is listed here together with other frameworks for reduction semantics, due to its close relationship with Rewriting Logic (translations both ways between the two frameworks have been provided [25, 3]), it could just as well have been classified as a structural framework. In fact it is a development of so-called *context systems* [19] where steps are specified much as in SOS, except that phrases may be contexts with multiple holes, and the actions that label the steps (the *effects*) may depend on actions to be provided by the holes (the *triggers*). A context may be thought of as an m-tuple of terms in n variables; there are operations, familiar from Lawvere's algebraic theories, for composing contexts sequentially (plugging the terms of one context into the holes of another) and in parallel (concatenating tuples of terms over the same variables), together with units and projections.

In TL, steps may affect the interfaces of contexts, and the steps themselves have a rich algebraic structure; see [13, 14] for the details. Here we shall merely introduce the notation of TL, and illustrate its use to express the operational semantics of a familiar fragment of CCS.

The conventional algebraic notation for a tile is $s \xrightarrow[b]{a} t$ (ignoring the label of the tile, for simplicity), where $s \longrightarrow t$ is a context rewrite step, a is the trigger of the step, and b is its effect. The tile requires that the variables of s are rewritten with a cumulative effect a.

For appropriate arguments, sequential composition of contexts is written $s;t$, with unit id, and parallel composition is written as $s \otimes t$. Duplicators are written ∇, and dischargers (or sinks) as ! (permuters are also provided). The operations satisfy all the axioms that one might expect.

Two tiles can be composed in parallel (using $\otimes$), vertically (using $\cdot$), or horizontally (using $*$), provided that their components have the appropriate types.

Finally, here are the tiles for some CCS constructs (where the variables x_1, x_2 are actually redundant, and are usually omitted):

$$\alpha.x_1 \xrightarrow[\alpha]{id} x_1 \tag{56}$$

The above rule may be read operationally as: the context prefixes the hole x_1 with the action α, and may become just the hole, emitting α as effect, without any trigger.

$$x_1 \mid x_2 \xrightarrow[\alpha]{\alpha \otimes id} x_1 \mid x_2 \tag{57}$$

$$x_1 \mid x_2 \xrightarrow[\alpha]{id \otimes \alpha} x_1 \mid x_2 \tag{58}$$

$$x_1 \mid x_2 \xrightarrow[\tau]{\alpha \otimes \bar{\alpha}} x_1 \mid x_2 \tag{59}$$

Of course, this simple kind of SOS example does not nearly exploit the full generality of the Tile Logic framework, which encompasses graph rewriting as well as rewriting logic and context systems.

4 Prototyping

The Maude implementation of Rewriting Logic (RL) [5] has several features that make it particularly attractive to use for prototyping operational semantics of programming languages. For instance, it provides meta-level functions for parsing, controlled rewriting, and pretty-printing; moreover, the Maude rewriting engine is highly efficient. Maude also supports Membership Algebra [24], which is an expressive framework for order-sorted algebraic specification.

Together with Christiano Braga at SRI International, the author has recently been developing a representation of Modular SOS (MSOS) [31, 33] in RL and implementing it in Maude; this involved first extending Maude with a new kind of conditional rule, using the Maude meta-level. (Presently, MSOS rules are translated manually to Maude rules, but later the translation is itself to be implemented using Maude meta-level facilities.)

The translation process transforms an MSOS specification into an SOS-like one [33]. MSOS rules are translated into Maude rules over configurations that have a syntactic and a semantic component. Label formulae are translated into equations dealing with the associated states. The MSOS of Action Notation [35] is being prototyped this way—when completely implemented, together with further meta-level functions for processing descriptions formulated in Action Semantics [29, 30], it should enable the prototyping of action-semantic descriptions of programming languages.

5 Conclusion

It is hoped that this survey of frameworks for the logical specification of operational semantics has provided a useful overview of much of the work in this area (apologies to those whose favourite frameworks have been omitted). It would be unwise to try to draw any definite conclusions on the basis of the remarks made here about the various frameworks: both the SOS and the reduction semantics approaches have their strengths, and are currently active areas of research—as is Tile Logic, which is strongly related to the structural approach, as well as to

Rewriting Logic. However, it is clear that logic has been found to be a particularly useful tool for specifying operational semantics, and appears to be preferred in practice to approaches based on abstract machines and interpreters.

Acknowledgements: Thanks to Christiano Braga, José Meseguer, and Carolyn Talcott for comments on a draft of this paper, and to José Meseguer for helpful discussions and support throughout the author's visit to SRI International. During the preparation of this paper 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. E. Astesiano and G. Reggio. SMoLCS driven concurrent calculi. In *TAPSOFT'87, Proc. Int. Joint Conf. on Theory and Practice of Software Development, Pisa*, volume 249 of *LNCS*. Springer-Verlag, 1987.
2. D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 119–129. ACM, 1992.
3. R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, Computer Science Lab., SRI International, July 1998.
4. 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.
5. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [22], pages 65–89.
6. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from http://www.brics.dk/Projects/CoFI.
7. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [6], Oct. 1998.
8. P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Proc. POPL'92*, pages 83–94. ACM, 1992.
9. P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, June 1996.
10. 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.
11. M. Felleisen and R. Hieb. The reveised report on the syntactic theories of sequential control and state. *Theoretical Comput. Sci.*, 102:235–271, 1992.
12. W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisumulation for core CML. *J. Functional Programming*, 8(5):447–451, 1998.
13. F. Gadducci and U. Montanari. Tiles, rewriting rules, and CCS. In Meseguer [22].
14. F. Gadducci and U. Montanari. The tile model. In *Proof, Language, and Interaction*. The MIT Press, 1999. To appear.

15. R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Robin Milner Festschrifft*. MIT Press, 1998. To appear.
16. A. S. A. Jeffrey. Semantics for core concurrent ML using computation types. In A. D. Gordon and A. J. Pitts, editors, *Higher Order Operational Techniques in Semantics, Proceedings*. Cambridge University Press, 1997.
17. 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.
18. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
19. K. G. Larsen and L. Xinxin. Compositionality through operational semantics of contexts. In *Proc. ICALP'90*, volume 443 of *LNCS*, pages 526–539. Springer-Verlag, 1990.
20. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay, editor, *Handbook of Philosophical Logic*, volume 6. Kluwer Academic Publishers, 1998. 'Also Technical Report SRI-CSL-93-05, SRI International, August 1993.
21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
22. J. Meseguer, editor. *First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996.
23. J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'96*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, 1996.
24. J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *LNCS*, pages 18–61. Springer-Verlag, 1998.
25. J. Meseguer and U. Montanari. Mapping tile logic into rewriting logic. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *LNCS*. Springer-Verlag, 1998.
26. 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.
27. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. ICALP'92*, volume 623 of *LNCS*, pages 685–695. Springer-Verlag, 1992.
28. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
29. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
30. 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.
31. 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`.
32. 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 [33].

33. 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 [32].
34. 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 [35].
35. 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 [34].
36. A. M. Pitts and J. R. X. Ross. Process calculus based upon evaluation to committed form. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'96*, volume 1119 of *LNCS*. Springer-Verlag, 1996.
37. G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Comput. Sci.*, 1:125–159, 1975.
38. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
39. C. Priami. *Enhanced Operational Semantics for Concurrency.* PhD thesis, Dipartimento di Informatica, Università degli Studi di Pisa, 1996.
40. 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.
41. J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Computer Science Dept., Cornell Univ., 1992. Tech. Rep. TR 92-1285.
42. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Dept. of Computer Science, Univ. of Edinburgh, 1992.
43. P. Sewell. From rewrite to bisimulation congruences. In *Proc. CONCUR'98*, volume 1466 of *LNCS*, pages 269–284. Springer-Verlag, 1998.

Recent BRICS Report Series Publications

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

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

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

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

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

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

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.