BRICS

Basic Research in Computer Science

BRICS RS-00-50 P. D. Mosses: Modularity in Meta-Languages

Modularity in Meta-Languages

Peter D. Mosses

BRICS Report Series **RS-00-50**

ISSN 0909-0878 **December 2000**

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/00/50/`

Modularity in Meta-Languages*

Peter D. Mosses

BRICS & Dept. of Computer Science, Univ. of Aarhus, Denmark

Abstract. A meta-language for semantics has a high degree of modularity when descriptions of individual language constructs can be formulated independently using it, and do not require reformulation when new constructs are added to the described language. The quest for modularity in semantic meta-languages has been going on for more than two decades. Here, most of the main meta-languages for operational, denotational, and hybrid styles of semantics are compared regarding their modularity. A simple bench-mark is used: describing the semantics of a pure functional language, then extending the described language with references, exceptions, and concurrency constructs. For each style of semantics, at least one of the considered meta-languages appears to provide a high degree of modularity.

1 Introduction

A meta-language for semantic description of programming languages may be regarded as having a high degree of modularity when it can be used to give descriptions of individual language constructs that can be formulated (and understood) independently of each other. As an illustrative bench-mark for assessing modularity, consider the extension of a pure functional language with new features: references, exceptions, and/or concurrent processes. Depending on whether or not the original description of the functional constructs requires significant reformulation when these new features are added to the language, the meta-language may be regarded as having a lower, resp. higher degree of modularity.

The quest for modularity in semantic meta-languages has been going on for more than two decades. The present author's contributions to it include: proposals for obtaining modularity in denotational semantics by use of particular styles of auxiliary notation [13], abstract semantic algebras [14], and action combinators [16]; the hybrid (denotational/operational) framework of action semantics [17, 18, 23, 31]; and, recently, a modular framework for (small- and big-step) structural operational semantics, called Modular SOS [20]. Moggi (provoked by the ad-hoc nature of the usual techniques for constructing domains in denotational semantics [15]) has proposed the use of monads and monad transformers [11], see also [9]; he has subsequently developed a more general framework based on translations between meta-languages [12]. Wansbrough and Hamer have proposed the hybrid framework of modular monadic action semantics [30].

* Presented at *LFM'2000, 2nd Workshop on Logical Frameworks and Meta-Languages*, Santa Barbara, California, USA, June 2000

Most of the major (and some minor) meta-languages that have been used for semantic description of programming languages are assessed below regarding the degree of modularity that they provide, using the illustrative bench-mark suggested above. Sect. 2 considers *operational semantics:* conventional SOS, Modular SOS, and Evaluation Contexts. Sect. 3 deals with *denotational semantics:* the conventional (Scott-Strachey) meta-language, monadic semantics, and Extensible Denotational Semantics. Sect. 4 assesses the approaches of Action Semantics and of Type-Theoretic Interpretation, which are hybrids of operational and denotational semantics.

Although it would be unwise to draw any definitive conclusions on the basis of the simplistic assessments reported below, it seems safe to claim that both Modular SOS and Action Semantics do perform particularly well regarding modularity—as does the monadic style of denotational semantics. In each case, achieving modularity was a stated aim of the design of the meta-language.

In fact modularity is just one of several "semantic engineering" aspects of meta-languages that may be crucial for their successful use in large-scale applications. Others include: the ease of writing, reading, and understanding descriptions; the possibility of generating compilers from language descriptions; and the feasibility of proving expected consequences of a semantic description. It appears that unfortunately, none of the current semantic meta-languages is completely satisfactory concerning all these semantic engineering aspects.

Caveat: Some of the finer details of the various meta-languages are ignored here, in an attempt to avoid distraction from the main points that are being made. Moreover, the author is not equally familiar with all the meta-languages considered, and some of the illustrative examples may be non-optimal, or even erroneous.

A Simple Bench-Mark

For the assessment of modularity of semantic meta-languages, it suffices to consider the description of a few typical constructs, forming tiny fragments of full programming languages. The constructs below have been selected mainly for their simplicity and familiarity, without (conscious) bias towards particular styles of semantic meta-language. They are (in some cases, simplified versions of) constructs taken from Standard ML and Concurrent ML.

Purely functional constructs

$$\mathit{exp} ::= \mathtt{if}\ \mathit{exp}\ \mathtt{then}\ \mathit{exp}\ \mathtt{else}\ \mathit{exp} \mid \mathit{exp}\ \mathtt{=}\ \mathit{exp}$$

Let the values $v \in V$ of expressions $\mathit{exp} \in \mathit{Exp}$ include both booleans $b \in B$ and numbers $n \in N$. Assume that only boolean values can be tested by if-expressions, and that both boolean and numerical values can be compared by equality expressions.

Constructs involving references

$$exp ::= \texttt{deref}\ exp \mid exp\ \texttt{:=}\ exp$$

Let expression values include locations $l \in L$, with dereferencing a location returning the value last assigned to it.

Constructs involving exceptions

$$exp ::= \texttt{fail} \mid \texttt{catch}\ exp\ \texttt{with}\ exp$$

Let exceptional expression values $x \in X$ include `fail`, with the most-recently started catch expression handling any `fail` that arises while evaluating its first sub-expression.

Constructs involving concurrency

$$exp ::= \texttt{synch} \mid \texttt{spawn}\ exp$$

Let `spawn` *exp* initiate the concurrent evaluation of the expression, and let any two synch-expressions match when they occur concurrently, each then evaluating to a null value ().

2 Operational Semantics

2.1 Conventional SOS

SOS (Structural Operational Semantics) is a particularly well-known meta-language for describing process algebras and programming languages. Following Plotkin [25], SOS has often been preferred to the more abstract framework of denotational semantics. Plotkin himself was concerned about the modularity of SOS:

> *As regards modularity we just hope that if we get the other things in a reasonable state, then current ideas for imposing modularity on specifications will prove useful.* [25, p. 64]

The so-called "big-step" or "natural semantics" style of SOS does not seem to differ significantly from Plotkin's original "small-step" style with respect to modularity, so it is not considered separately here. Note also that the recent work of Plotkin and Turi, aiming to reconcile operational and denotational semantics [28], does not address the kind of modularity considered here.

Suppose that a pure functional language, including the functional constructs of the bench-mark indicated in Sect. 1, is to be described using conventional SOS. The configurations $\gamma \in \Gamma$ of a straightforward SOS for it would be simply the "value-added" syntax obtained by extending the constructors for expressions with the computed values as constants (or merely the original syntax, if it already includes suitable canonical terms representing computed values). For instance, if the boolean values tt and $f\!f$ are not directly expressible, one should add to the grammar for expressions $exp \in Exp$ the following extra productions:

$$exp ::= tt \mid f\!f$$

and take Γ to be Exp (which is of course different from taking $\Gamma = Exp \cup \{tt, f\!f\}$ without extending the grammar for expressions). Similarly, one may evaluate numerals to the corresponding abstract numbers, which would then also have to be admitted as components of expressions.

The transition relation to be specified by the SOS rules may be written $exp \rightarrow exp'$, and values v are terminal configurations.[1] The rules for transitions are then specified as follows:

$$\frac{exp_1 \longrightarrow exp'_1}{\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3 \longrightarrow \texttt{if } exp'_1 \texttt{ then } exp_2 \texttt{ else } exp_3} \tag{1}$$

$$\texttt{if } tt \texttt{ then } exp_2 \texttt{ else } exp_3 \longrightarrow exp_2 \tag{2}$$

$$\texttt{if } f\!f \texttt{ then } exp_2 \texttt{ else } exp_3 \longrightarrow exp_3 \tag{3}$$

$$\frac{exp_1 \longrightarrow exp'_1}{exp_1 \texttt{ = } exp_2 \longrightarrow exp'_1 \texttt{ = } exp_2} \qquad \frac{exp_2 \longrightarrow exp'_2}{v_1 \texttt{ = } exp_2 \longrightarrow v_1 \texttt{ = } exp'_2} \tag{4}$$

$$v \texttt{ = } v \longrightarrow tt \qquad v_1 \texttt{ = } v_2 \longrightarrow f\!f \text{ if } v_1 \neq v_2 \tag{5}$$

If the functional language is to be extended to a language with references, including dereferencing and assignment expressions such as those indicated in Sect. 1, stores $s \in S$ would be added as an extra component to configurations, and the transition relation would become $exp, s \rightarrow exp', s'$ (see e.g. [1]). All the original rules for the functional language are unfortunately now *invalid*, and would have to be *reformulated* thus:

$$\frac{exp_1, s \longrightarrow exp'_1, s'}{\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3, s \longrightarrow \texttt{if } exp'_1 \texttt{ then } exp_2 \texttt{ else } exp_3, s'} \tag{6}$$

$$\texttt{if } tt \texttt{ then } exp_2 \texttt{ else } exp_3, s \longrightarrow exp_2, s \tag{7}$$

$$\texttt{if } f\!f \texttt{ then } exp_2 \texttt{ else } exp_3, s \longrightarrow exp_3, s \tag{8}$$

$$\frac{exp_1, s \longrightarrow exp'_1, s'}{exp_1 \texttt{ = } exp_2, s \longrightarrow exp'_1 \texttt{ = } exp_2, s'} \qquad \frac{exp_2, s \longrightarrow exp'_2, s'}{v_1 \texttt{ = } exp_2, s \longrightarrow v_1 \texttt{ = } exp'_2, s'} \tag{9}$$

$$v \texttt{ = } v, s \longrightarrow tt, s \qquad v_1 \texttt{ = } v_2, s \longrightarrow f\!f, s \text{ if } v_1 \neq v_2 \tag{10}$$

[1] If the expressions include binding constructs, their semantics may be described using substitution. Alternatively explicit environments $\rho \in Env$ may be introduced, writing $\rho \vdash exp \rightarrow exp'$ for the (relative) transition relation [25, p. 72].

The necessity of such a reformulation confirms that conventional SOS indeed has some problems regarding modularity. After the reformulation, the rules for dereferencing and assignment can now be added:

$$\frac{exp, s \longrightarrow exp', s'}{\texttt{deref}\ exp, s \longrightarrow \texttt{deref}\ exp', s'} \tag{11}$$

$$\texttt{deref}\ l, s \longrightarrow v, s \text{ if } s(l) = v \tag{12}$$

$$\frac{exp_1, s \longrightarrow exp_1', s'}{exp_1\ \texttt{:=}\ exp_2, s \longrightarrow exp_1'\ \texttt{:=}\ exp_2, s'} \qquad \frac{exp_2, s \longrightarrow exp_2', s'}{l_1\ \texttt{:=}\ exp_2, s \longrightarrow l_1\ \texttt{:=}\ exp_2', s'} \tag{13}$$

$$l_1\ \texttt{:=}\ v_2, s \longrightarrow v_2, s[l_1 \mapsto v_2] \tag{14}$$

On the other hand, if the functional language is to be extended to a language where expressions may raise (and handle) exceptions, no extra components are needed in configurations, and the original rules for functional constructs may be retained—provided that the raised exceptions computed by expressions are kept separate from the ordinary computed values. New rules (in fact just axioms in the small-step style illustrated here) need to be added to specify how each functional construct propagates exceptions, which might be tedious for a large language; but for the functional constructs included in our bench-mark, we need add only:

$$\texttt{if}\ x_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \longrightarrow x_1 \tag{15}$$

$$x_1\ \texttt{=}\ exp_2 \longrightarrow x_1 \qquad v_1\ \texttt{=}\ x_2 \longrightarrow x_2 \tag{16}$$

(where x_1, x_2 range over X, the raised exception values, assumed to include `fail`). The rules for the new constructs themselves are:

$$\frac{exp_1 \longrightarrow exp_1'}{\texttt{catch}\ exp_1\ \texttt{with}\ exp_2 \longrightarrow \texttt{catch}\ exp_1'\ \texttt{with}\ exp_2} \tag{17}$$

$$\texttt{catch fail with}\ exp_2 \longrightarrow exp_2 \qquad \texttt{catch}\ v_1\ \texttt{with}\ exp_2 \longrightarrow v_1 \tag{18}$$

$$\texttt{catch}\ x_1\ \texttt{with}\ exp_2 \longrightarrow x_1 \text{ if } x_1 \neq \texttt{fail} \tag{19}$$

Thus the modularity of SOS may be considered satisfactory regarding the independence of the descriptions of normal and exceptional evaluation.

Finally, if the functional language is to be extended to a language with CML-style concurrency constructs, transitions may be labelled by signals $\alpha \in A = \{\texttt{synch}\} \cup \{\texttt{spawn}\ exp \mid exp \in Exp\} \cup \{\tau\}$ for synchronization and process-spawning, the transition relation then being written $exp \xrightarrow{\alpha} exp'$ (see e.g. [1]). Here, as when adding references, all the original rules for the functional language again have to be reformulated:

$$\frac{exp_1 \xrightarrow{\alpha} exp_1'}{\texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \xrightarrow{\alpha} \texttt{if}\ exp_1'\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3} \tag{20}$$

$$\texttt{if}\ tt\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \xrightarrow{\tau} exp_2 \tag{21}$$

$$\texttt{if}\ \mathit{ff}\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \xrightarrow{\tau} exp_3 \tag{22}$$

$$\frac{exp_1 \xrightarrow{\alpha} exp_1'}{exp_1\ \texttt{=}\ exp_2 \xrightarrow{\alpha} exp_1'\ \texttt{=}\ exp_2} \qquad \frac{exp_2 \xrightarrow{\alpha} exp_2'}{v_1\ \texttt{=}\ exp_2 \xrightarrow{\alpha} v_1\ \texttt{=}\ exp_2'} \tag{23}$$

$$v\ \texttt{=}\ v \xrightarrow{\tau} tt \qquad v_1\ \texttt{=}\ v_2 \xrightarrow{\tau} \mathit{ff}\ \text{if}\ v_1 \neq v_2 \tag{24}$$

(Letting the signals be terminal configurations and putting the computed values as labels [5] would require even greater reformulation.) The rules for the concurrency constructs may now be added:

$$\texttt{synch} \xrightarrow{\texttt{synch}} () \tag{25}$$

$$\texttt{spawn}\ exp \xrightarrow{\texttt{spawn}\ exp} () \tag{26}$$

$$\frac{proc_1 \xrightarrow{\alpha} proc_1'}{proc_1 \parallel proc_2 \xrightarrow{\alpha} proc_1' \parallel proc_2} \qquad \frac{proc_2 \xrightarrow{\alpha} proc_2'}{proc_1 \parallel proc_2 \xrightarrow{\alpha} proc_1 \parallel proc_2'} \tag{27}$$

$$\frac{proc_1 \xrightarrow{\texttt{synch}} proc_1' \qquad proc_2 \xrightarrow{\texttt{synch}} proc_2'}{proc_1 \parallel proc_2 \xrightarrow{\tau} proc_1' \parallel proc_2'} \tag{28}$$

$$\frac{exp \xrightarrow{\texttt{spawn}\ exp''} exp'}{exp \xrightarrow{\tau} exp' \parallel exp''} \tag{29}$$

The overall conclusion, based on our simple bench-mark, is that conventional SOS has a rather low degree of modularity. This conclusion is confirmed by the examples of SOS that are given in various textbooks on semantics (e.g. [24]).

2.2 Modular SOS

As its name suggests, the Modular SOS (MSOS) meta-language [19, 20] is a variant of SOS that provides significantly greater modularity than the conventional (small- or big-step) framework. The development of MSOS was prompted by the success of the monadic approach to obtaining modularity in denotational semantics, and by the lack of a satisfactory answer to the question of whether or not something similar might be possible for operational semantics.

The key idea of MSOS is to restrict configurations to pure (value-added) syntax, putting all auxiliary information (stores, environments, signals, etc.) in the *labels* on transitions (in fact the set of labels naturally forms a category, and labels on adjacent transitions are required to be composable). Thus the transition relation for expressions in MSOS is *always* of the form $exp \xrightarrow{\alpha} exp'$, where α ranges over the labels. By treating labels as an abstract datatype whose components can be accessed and changed independently of each other, MSOS rules never have to be reformulated when new components are added.

In an MSOS for a functional language, environments may be included in the labels, or substitution may be used in the rules for binding constructs—the formulation of the rules for constructs not concerned with binding is unaffected by the choice. Thus for the bench-mark functional constructs, the MSOS rules would always be as follows:

$$\frac{\mathit{exp}_1 \xrightarrow{\alpha} \mathit{exp}'_1}{\texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \xrightarrow{\alpha} \texttt{if}\ \mathit{exp}'_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3} \tag{30}$$

$$\texttt{if}\ \mathit{tt}\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \xrightarrow{\iota} \mathit{exp}_2 \tag{31}$$

$$\texttt{if}\ \mathit{ff}\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \xrightarrow{\iota} \mathit{exp}_3 \tag{32}$$

$$\frac{\mathit{exp}_1 \xrightarrow{\alpha} \mathit{exp}'_1}{\mathit{exp}_1\ \texttt{=}\ \mathit{exp}_2 \xrightarrow{\alpha} \mathit{exp}'_1\ \texttt{=}\ \mathit{exp}_2} \qquad \frac{\mathit{exp}_2 \xrightarrow{\alpha} \mathit{exp}'_2}{v_1\ \texttt{=}\ \mathit{exp}_2 \xrightarrow{\alpha} v_1\ \texttt{=}\ \mathit{exp}'_2} \tag{33}$$

$$v\ \texttt{=}\ v \xrightarrow{\iota} \mathit{tt} \qquad v_1\ \texttt{=}\ v_2 \xrightarrow{\iota} \mathit{ff}\ \text{if}\ v_1 \neq v_2 \tag{34}$$

The variable α ranges over all the arrows of the category of labels, about which no assumptions need be made here. The use of α in the rules above specifies full propagation of all information processing between steps of evaluating compound expressions and those of their sub-expressions. In contrast, the variable ι ranges only over the *identity arrows* of the label category, which always label inherently unobservable transitions, not affecting any information being processed.

An extension with references would involve labels including (pairs of) stores, but no reformulation of the rules for functional constructs would be needed: the variables α and ι in the original rules simply range over the new category of labels (see e.g. [21]). In fact adding stores to labels (using a product-forming "label transformer") preserves not only the values computed by expressions, but also the computations themselves [19].

Let $\mathit{get}_s(\alpha)$ return the first component of the pair of stores in α, and let $\mathit{set}_s(\alpha, s')$ return a label whose components are the same as those of α, except that the second component of the pair of stores is now s'. The MSOS rules for the bench-mark constructs involving references are as follows:

$$\frac{\mathit{exp} \xrightarrow{\alpha} \mathit{exp}'}{\texttt{deref}\ \mathit{exp} \xrightarrow{\alpha} \texttt{deref}\ \mathit{exp}'} \tag{35}$$

$$\texttt{deref}\ l \xrightarrow{\iota} v\ \text{if}\ \mathit{get}_s(\iota)(l) = v \tag{36}$$

$$\frac{\mathit{exp}_1 \xrightarrow{\alpha} \mathit{exp}'_1}{\mathit{exp}_1\ \texttt{:=}\ \mathit{exp}_2 \xrightarrow{\alpha} \mathit{exp}'_1\ \texttt{:=}\ \mathit{exp}_2} \qquad \frac{\mathit{exp}_2 \xrightarrow{\alpha} \mathit{exp}'_2}{l_1\ \texttt{:=}\ \mathit{exp}_2 \xrightarrow{\alpha} l_1\ \texttt{:=}\ \mathit{exp}'_2} \tag{37}$$

$$l_1\ \texttt{:=}\ v_2 \xrightarrow{\alpha} v_2\ \text{if}\ \alpha = \mathit{set}_s(\iota, \mathit{get}_s(\iota)[l_1 \mapsto v_2]) \tag{38}$$

The applications above of get_s and set_s remain valid when further components of labels are added.

As with conventional SOS, an extension with exceptions would not require any reformulation of the rules for functional constructs, but it would require an additional axiom or two for each:

$$\texttt{if}\ x_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \xrightarrow{\iota} x_1 \tag{39}$$

$$x_1\ \texttt{=}\ exp_2 \xrightarrow{\iota} x_1 \qquad v_1\ \texttt{=}\ x_2 \xrightarrow{\iota} x_2 \tag{40}$$

Finally, since the transitions in MSOS are already labelled, it is unsurprising that the original rules for the functional constructs can be retained also when extending with concurrency constructs. Labels need an extra component whose values represent *sequences* of synchronization and process spawning signals; let $get_a(\alpha)$ return the sequence of signals in α, and let $set_a(\alpha, a_1 \dots a_n)$ return a label whose components are the same as those of α, except that the sequence of signals is now $a_1 \dots a_n$. As with a conventional SOS, auxiliary configurations of concurrent processes $proc \in Proc$ are required; a single expression exp is a single process (but processes are not allowed as sub-expressions). The MSOS rules for the bench-mark constructs involving references are then as follows:

$$\texttt{synch} \xrightarrow{\alpha} ()\ \text{if}\ \alpha = set_a(\iota, \texttt{synch}) \tag{41}$$

$$\texttt{spawn}\ exp \xrightarrow{\alpha} ()\ \text{if}\ \alpha = set_a(\iota, (\texttt{spawn}\ exp)) \tag{42}$$

$$\frac{proc_1 \xrightarrow{\alpha} proc_1'}{proc_1 \parallel proc_2 \xrightarrow{\alpha} proc_1' \parallel proc_2} \qquad \frac{proc_2 \xrightarrow{\alpha} proc_2'}{proc_1 \parallel proc_2 \xrightarrow{\alpha} proc_1 \parallel proc_2'} \tag{43}$$

$$\frac{proc_1 \xrightarrow{\alpha_1} proc_1' \qquad proc_2 \xrightarrow{\alpha_2} proc_2'}{proc_1 \parallel proc_2 \xrightarrow{\iota} proc_1' \parallel proc_2'}\ \text{if}\ \begin{array}{l}\alpha_1 = set_a(\iota, \texttt{synch})\\ \alpha_2 = set_a(\iota, \texttt{synch})\end{array}\ \text{and} \tag{44}$$

$$\frac{exp \xrightarrow{\alpha} exp'}{exp \xrightarrow{\iota} exp' \parallel exp''}\ \text{if}\ \alpha = set_a(\iota, (\texttt{spawn}\ exp'')) \tag{45}$$

Full details of the extensions of a functional language with references and/or concurrency are provided in [21], together with a comparison of that MSOS with other published descriptions of comparable languages.

2.3 Evaluation Contexts

A popular alternative to conventional SOS is to model steps of computations as reductions in a term rewriting system. The terms are formed from the syntax of the described language, together with (irreducible) terms representing computed values and auxiliary information (such as stores). Reductions $exp \to exp'$ are restricted to occur only in *evaluation contexts* [3, 4, 32], written $E[\,]$, the form of which is specified by context-free grammars. The empty context is denoted by $[\,]$.

A semantic description of functional constructs in this meta-language corresponds closely to a description in conventional small-step SOS (using substitution, rather than environments, to deal with bindings). For the bench-mark constructs, evaluation contexts are specified thus:

$$E ::= [\,] \mid \texttt{if}\ E\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \mid E\ \texttt{=}\ \mathit{exp} \mid v\ \texttt{=}\ E$$

The reduction rules are as follows:

$$\texttt{if}\ \mathit{tt}\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \rightarrow \mathit{exp}_2 \tag{46}$$

$$\texttt{if}\ \mathit{ff}\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \rightarrow \mathit{exp}_3 \tag{47}$$

$$v\ \texttt{=}\ v \rightarrow \mathit{tt} \qquad v_1\ \texttt{=}\ v_2 \rightarrow \mathit{ff}\ \text{if}\ v_1 \neq v_2 \tag{48}$$

$$E[\mathit{exp}] \longrightarrow E[\mathit{exp}']\ \text{iff}\ \mathit{exp} \rightarrow \mathit{exp}' \tag{49}$$

Each alternative of the above grammar for evaluation contexts may be regarded as abbreviating an inference rule of the small-step SOS description given in Sect. 2.1, and each reduction rule $\mathit{exp} \rightarrow \mathit{exp}'$ corresponds exactly to an SOS axiom.

If the functional language is to be extended with references, a syntactic representation of stores may be added as a component to the (top-level) evaluation contexts:

$$P ::= E, s$$

The reduction rules for the functional constructs do not need any reformulation, although their lifting to computation steps has to be extended to the new contexts:

$$E[\mathit{exp}], s \longrightarrow E[\mathit{exp}'], s\ \text{iff}\ \mathit{exp} \rightarrow \mathit{exp}' \tag{50}$$

Assignment and dereferencing may then be described by extending the grammar of the original evaluation contexts:

$$E ::= \texttt{deref}\ E \mid E\ \texttt{:=}\ \mathit{exp} \mid l\ \texttt{:=}\ E$$

and adding new *computation* rules of the form $E[\mathit{exp}], s \rightarrow E[\mathit{exp}'], s'$:

$$E[\texttt{deref}\ l], s \longrightarrow E[v], s\ \text{if}\ s(l) = v \tag{51}$$

$$E[l_1\ \texttt{:=}\ v_2], s \longrightarrow E[v_2], s[l_1 \mapsto v_2] \tag{52}$$

An extension of a functional language with exceptions may be described by first adding further evaluation contexts:

$$\begin{aligned} &E ::= \texttt{catch}\ E\ \texttt{with}\ \mathit{exp} \\ &C ::= [\,] \mid \texttt{if}\ C\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \mid C\ \texttt{=}\ \mathit{exp} \mid v\ \texttt{=}\ C \end{aligned}$$

The following rules allow the reduction of a raised exception together with its innermost exception-handling context:

$$\texttt{catch}\ U[\texttt{fail}]\ \texttt{with}\ \mathit{exp} \rightarrow \mathit{exp} \tag{53}$$

$$\texttt{catch}\ v\ \texttt{with}\ exp \to v \tag{54}$$

Again, no reformulation of the original reduction rules is required. Notice however that it may be quite tedious to define contexts like C which are like other contexts except for omitting a single construct: the alternatives for C above correspond closely to the axioms needed in a (small-step) SOS or MSOS for propagating exceptions through functional constructs.

Finally, an extension with concurrency constructs may follow Reppy [26, 27] by exploiting evaluation contexts to match process expressions with subexpressions whose evaluation requires synchronization or process-spawning. As in SOS, auxiliary syntax for concurrent processes $proc \in Proc$ is required:

$$proc ::= exp \mid proc \parallel proc$$

Evaluation contexts for sets of processes need introducing:

$$P ::= E \mid P \parallel proc \mid proc \parallel P$$

No reformulation of the original reduction rules for expression evaluation is required. The following additional rules match the (otherwise irreducible) terms where a synchronization or spawning expression is to be evaluated, and lift reduction to computation steps for complete systems of processes:

$$P_1[\texttt{synch}] \parallel P_2[\texttt{synch}] \to P_1[()] \parallel P_2[()] \tag{55}$$

$$E[\texttt{spawn}\ exp] \to E[()] \parallel exp \tag{56}$$

$$P[proc] \longrightarrow P[proc'] \text{ iff } proc \to proc' \tag{57}$$

A drawback of this approach, compared to conventional SOS, is the lack of a basis for bisimulation equivalence of processes [5]. Nevertheless, according to our bench-mark, the modularity when using evaluation contexts appears to be just as good as when using MSOS.

3 Denotational Semantics

3.1 Conventional Denotational Semantics

It is well-known [3, 9, 11, 13–16, 23, 30] that the straightforward use of λ-notation (with fixed interpretation of λ-abstraction, application, and composition in domains of functions) leads to poor modularity in denotational semantics. This is merely confirmed by our bench-mark, as follows.

A conventional denotational semantics for a functional language takes denotations of expressions to be in $Env \to V$, where environments $\rho \in Env$ provide the values of free variables of expressions, and V is the domain of values computed by expressions. Thus:

$$E : Exp \to Env \to V$$

$$E[\![\texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3]\!] = \tag{58}$$
$$\lambda\rho.(E[\![exp_1]\!]\rho|B \to E[\![exp_2]\!]\rho, E[\![exp_3]\!]\rho)$$

$$E[\![exp_1\ \texttt{=}\ exp_2]\!] = \lambda\rho.(E[\![exp_1]\!]\rho = E[\![exp_2]\!]\rho) \tag{59}$$

An extension with references would involve changing the domain of denotations to $Env \to (S \to (V \times S))$, and a complete reformulation of the semantic equations for functional constructs would be required, as follows:

$$E : Exp \to Env \to S \to V \times S$$

$$E[\![\mathtt{if}\ exp_1\ \mathtt{then}\ exp_2\ \mathtt{else}\ exp_3]\!] = \tag{60}$$
$$\lambda\rho.\lambda s.(\lambda(v_1, s_1).v_1|B \to E[\![exp_2]\!]\rho s_1, E[\![exp_3]\!]\rho s_1)(E[\![exp_1]\!]\rho s)$$

$$E[\![exp_1\ \mathtt{=}\ exp_2]\!] = \tag{61}$$
$$\lambda\rho.\lambda s.(\lambda(v_1, s_1).(\lambda(v_2, s_2).v_1 = v_2)(E[\![exp_2]\!]\rho s_1))(E[\![exp_1]\!]\rho s)$$

An extension with exceptions could either replace $Env \to V$ by $Env \to (V + X)$, where X is the domain of raised exception values, or switch from the "direct" to the "continuation" style by taking denotations in $Env \to K \to V$ where $K = V \to V$; in both cases, complete reformulation of the equations for functional constructs would again be required:

$$E : Exp \to Env \to K \to V$$

$$E[\![\mathtt{if}\ exp_1\ \mathtt{then}\ exp_2\ \mathtt{else}\ exp_3]\!] = \tag{62}$$
$$\lambda\rho.\lambda\kappa.E[\![exp_1]\!]\rho(\lambda v.v|B \to E[\![exp_2]\!]\rho\kappa, E[\![exp_3]\!]\rho\kappa)$$

$$E[\![exp_1\ \mathtt{=}\ exp_2]\!] = \tag{63}$$
$$\lambda\rho.\lambda\kappa.E[\![exp_1]\!]\rho(\lambda v_1.E[\![exp_2]\!]\rho(\lambda v_2.\kappa(v_1 = v_2)))$$

Finally, an extension with concurrency constructs would require denotations to be in $Env \to \mathcal{P}(R)$, where R is some domain of "resumptions", and a corresponding reformulation of the original equations for functional constructs (not illustrated here).

Thus the modularity of conventional denotational semantics is as *poor* as it could possibly be, according to our bench-mark.

3.2 Monadic Denotational Semantics

The use of a monadic meta-language in denotational semantics has been proposed by Moggi [11], and adopted by Liang and Hudak [9], among others.

The key idea of monadic semantics is to avoid any dependency on the structure of denotations when giving the equations that define the semantic functions. This is achieved by regarding denotations as elements of an abstract computation type, equipped with operations for composing computations (or functions from computed values to computations) and for forming trivial computations from computed values, satisfying the laws of monads. Particular monads may provide further operations on computations and values.

A monadic semantics provides a particular definition of a monad of computations, defining for each type of values τ the domain $T\tau$ of computations of values of type τ, and defining composition and other operations using conventional λ-notation. When the described language is changed or extended, the semantic equations generally do not need any reformulation at all, even though the definition of the domain of computations changes. This already provides a substantial degree of modularity in denotational semantics.

However, the definitions of composition and of the other operations provided by the monad generally have to be completely reformulated each time the domain of computations is changed. To avoid this, Moggi proposed to construct monads *incrementally*: transforming a monad T with domains of computation $T\tau$ into a richer monad T' whose computations $T'\tau$ are defined in terms of $T\tau$, and whose composition is defined in terms of that of T, and then adding new operations associated with the added structure. Ideally, all the operations provided by T would automatically be lifted to operations on T', avoiding the need for explicit reformulation; but this turned out not to be possible, in general.

Let us check the claimed modularity of monadic semantics using our benchmark. To describe a pure functional language, one may start with the trivial monad I, where the computations are simply values. This is provided that binding constructs are treated as functions, using higher-order abstract syntax; an alternative, closer to conventional denotational semantics, is to construct an environment monad, where the domain of computations $T\tau$ of values of type τ is $Env \to \tau$, and with operations both for computing the current environment and for setting it. The choice of monad does not affect the semantic equations for constructs not involving binding, which are as follows:

$$E : Exp \to Env \to T\,V$$

$$E[\![\texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3]\!] = \tag{64}$$

$$\text{let } v = E[\![exp_1]\!] \text{ in } (v|B \to E[\![exp_2]\!], E[\![exp_3]\!])$$

$$E[\![exp_1\ \texttt{=}\ exp_2]\!] = \tag{65}$$

$$\text{let } v_1 = E[\![exp_1]\!] \text{ in let } v_2 = E[\![exp_2]\!] \text{ in } [v_1 = v_2]$$

In Moggi's notation, let $x = c_1$ in c_2 is sequential composition of the computation c_1 of value x followed by computation c_2, which may depend on x. The injection of values as trivial computations is written $[v]$.

For the extension with references, a side-effect monad T' is needed, with computations of the form $T'\tau = S \to (T\tau \times S)$, and with operations $update : L \times V \to T()$ for assignment and $lookup : L \to T\,V$ for dereferencing. When an environment monad was used for the functional language, the side-effect monad constructor needs to be applied before the environment constructor, but the side-effect operations can then be lifted automatically to the environment monad, so no explicit reformulation is required at all before adding:

$$E[\![\texttt{deref}\ exp]\!] = \text{let } l = E[\![exp]\!] \text{ in } lookup(l) \tag{66}$$

$$E[\![exp_1\ \texttt{:=}\ exp_2]\!] = \tag{67}$$

$$\text{let } l_1 = E[\![exp_1]\!] \text{ in let } v_2 = E[\![exp_2]\!] \text{ in } update(l_1, v_2)$$

The extension to exceptions may use the simple exception monad constructor that maps $T\tau$ to $T(\tau + X)$ for some domain of raised exception values X, providing $raise : X \to T\tau$ and $handle : X \times T\tau \times T\tau \to T\tau$ (the alternative is to use the more complex continuation monad constructor.) As with side-effects, this constructor should be applied before using an environment monad constructor, but the operations can again be lifted automatically.

$$E[\![\texttt{fail}]\!] = \mathit{raise}(\texttt{fail}) \tag{68}$$

$$E[\![\texttt{catch}\ \mathit{exp}_1\ \texttt{with}\ \mathit{exp}_2]\!] = \mathit{handle}(\texttt{fail}, E[\![\mathit{exp}_1]\!], E[\![\mathit{exp}_2]\!]) \tag{69}$$

The final extension incorporated in our bench-mark concerns concurrency constructs, which involve nondeterminism. Although there is no monad constructor for adding nondeterminism to side-effect (or even environment) monads, one may start from a monad for nondeterminism, instead of from the trivial monad. Then a monad constructor for resumptions may be applied. In principle, one may proceed to describe the desired concurrency constructs in terms of the operations provided by the monads (not illustrated here), without making any changes to the description of the functional language.

Thus it appears that the modularity of monadic denotational semantics is as *good* as it could possibly be, according to our bench-mark. Nevertheless, Moggi has recently proposed a more general framework based on composing *translations between meta-languages* [12]. The problem of lifting translations of some operations through other translations is dealt with by allowing redefinition of those operations to be expressed. This framework uses LF as a meta-meta-language, and emphasizes the type-theoretic aspects of monadic denotational semantics.

3.3 Extensible Denotational Semantics

Cartwright and Felleisen [3] have proposed the use of an operationally-motivated style of denotational semantics, in the interests of modularity. As with monadic denotational semantics, a distinction is made between the domain of computed values V and that of computations C, but here, C is defined to be (roughly) $V + ((V \to C) \times A)$, where only the domains V and A (of "actions") depend on the language described. Computations have to be composed using a special function, $\mathit{handler} : C \to (V \to C) \to C$; however, it does not satisfy all the monad laws. The semantics of a complete program is given by applying a further function, $\mathit{admin} : (C \times R) \to ((V \times E) \times R)$, where R is a language-dependent domain of "resources", and E is a fixed domain of "errors".

To describe a functional language using this meta-language, the domains V, A, and R are defined appropriately, and semantic equations for expressions are given using *handler* to combine the denotations of sub-expressions. An extension with references requires changes to the definitions of V, A, and R, as well as some equations defining *admin* on the new elements of A, but the semantic equations for functional constructs do not have to be reformulated at all. An extension with exceptions (or even with first-class continuations) likewise does not require any reformulation of the original semantic equations. The notation used in the semantic equations is comparable to that used in monadic semantics, and so it is not illustrated here.

Unfortunately, however, it appears that an extension with concurrency constructs is simply not possible in this meta-language, since the definition of C does not allow for the possibility of nondeterminism. Thus although it is conceivable that the meta-language could be relaxed to allow such changes, we must here conclude that the modularity of this approach, while better than that of conventional denotational semantics, falls short of that found in monadic denotational semantics.

4 Hybrid Approaches

In this section we consider meta-languages that combine the operational and denotational approaches.

4.1 Action Semantics

The present author, in collaboration with Watt, has proposed a framework called action semantics [17, 18, 23, 31]. The meta-language of action semantics includes a rich notation for so-called *actions*, which are used as denotations of programming constructs in much the same way as abstract computations are used in monadic denotational semantics—in particular, action notation includes a combinator that corresponds closely to composition of (functions from values to) computations in monads. However, there are also some significant differences between monadic and action semantics:

– The intended interpretation of action notation is defined *operationally*, using a small-step SOS, and data-types are specified algebraically [17], so action semantics does not involve λ-notation or domains at all.
– The various so-called *facets* of action notation support a *particular* combination of major computational concepts (data flow, control flow, scopes of bindings, effects on storage, and distributed processing, without bias towards any particular programming paradigm). In principle, action notation could be extended with new facets, and the existing facets could be replaced by alternative versions with different primitives and combinators; but in practice, action-semantic descriptions have used action notation as originally defined.
– Although some laws have been provided for action notation, its equational theory is weak. (However, a revised version of action notation has recently been proposed [8], with a significantly simpler kernel and a much stronger theory.)

An action semantics for a pure functional language uses actions with only data flow, control flow, and binding facets as the denotations of expressions. When the functional language is extended with references, exceptions, and/or concurrency constructs, these actions may have further facets, but the action notation used in describing the functional constructs does not require any reformulation at all.

$$
\begin{aligned}
&\mathit{evaluate} : \mathit{Exp} \to \mathit{Action} \\
&\mathit{evaluate}[\![\texttt{if } \mathit{exp}_1 \texttt{ then } \mathit{exp}_2 \texttt{ else } \mathit{exp}_3]\!] = && (70)\\
&\quad \mathit{evaluate\ exp}_1 \mathit{\ then} \\
&\quad ((\mathit{check\ (it\ is\ true)\ then\ evaluate\ exp}_2) \mathit{\ or} \\
&\quad (\mathit{check\ not\ (it\ is\ true)\ then\ evaluate\ exp}_3)) \\
&\mathit{evaluate}[\![\mathit{exp}_1 \texttt{ = } \mathit{exp}_2]\!] = && (71)\\
&\quad (\mathit{evaluate\ exp}_1 \mathit{\ and\ then\ evaluate\ exp}_2) \mathit{\ then} \\
&\quad \mathit{give\ (given\ value\#1\ is\ given\ value\#2)} \\
&\mathit{evaluate}[\![\texttt{deref } \mathit{exp}]\!] = && (72)\\
&\quad \mathit{evaluate\ exp\ then\ give\ the\ value\ stored\ in\ the\ given\ cell} \\
&\mathit{evaluate}[\![\mathit{exp}_1 \texttt{ := } \mathit{exp}_2]\!] = && (73)\\
&\quad (\mathit{evaluate\ exp}_1 \mathit{\ and\ then\ evaluate\ exp}_2) \mathit{\ then} \\
&\quad \mathit{store\ given\ value\#2\ in\ given\ cell\#1} \\
&\mathit{evaluate}[\![\texttt{fail}]\!] = \mathit{escape\ with\ } \texttt{fail} && (74)\\
&\mathit{evaluate}[\![\texttt{catch } \mathit{exp}_1 \texttt{ with } \mathit{exp}_2]\!] = && (75)\\
&\quad \mathit{evaluate\ exp}_1 \mathit{\ trap} \\
&\quad ((\mathit{check\ (it\ is\ } \texttt{fail}) \mathit{\ and\ then\ evaluate\ exp}_2) \mathit{\ or} \\
&\quad (\mathit{check\ not\ (it\ is\ } \texttt{fail}) \mathit{\ and\ then\ escape}))
\end{aligned}
$$

See [22] for an action semantics of process synchronization and spawning, showing that adding such constructs to a functional language does not involve changes to the original description. Thus according to our simple bench-mark, the degree of modularity of action semantics is as good as with MSOS and monadic semantics.

Wansbrough and Hamer, however, were not satisfied with the original definition of action notation: it was a monolithic conventional SOS, which would generally require major reformulation when adding new facets. They proposed [30] instead to define action notation incrementally by a monadic semantics, using monad constructors to combine facets. Wansbrough [29] gave a monadic semantics for much of the original action notation—omitting however most of the facet concerned with concurrent processes—and showed how a facet supporting continuations could be added to action notation. Wansbrough and Hamer dubbed their framework "modular monadic action semantics". The description of the bench-mark constructs in this approach would be exactly the same as with the original action semantics, except that a description of the concurrency constructs could not be included here without a major enhancement of the monad supporting the corresponding facet.

Stimulated by the apparent success of monadic semantics in giving a modular definition of (most of) action notation, the present author developed Modular SOS (MSOS, see Sect. 2.2), and has recently given an MSOS description of (the complete) action notation. It is however still unclear how to extend the MSOS of action notation with support for continuations (a mixture of evaluation contexts and MSOS might be useful here).

In any case, action notation provides good support for modular descriptions of most conventional high-level programming languages, regardless of how it is itself defined.

4.2 Type-Theoretic Interpretations

The last meta-language considered here is that used by Harper and Stone to give an alternative definition of Standard ML [6], based on type-theoretic interpretation. The original definition [10] was given using the big-step style of SOS (the "natural semantics" approach developed by Kahn [7]), but it resorted to "conventions" concerning the propagation of exceptions and states in order to achieve (reasonable) conciseness.

Type-theoretic interpretation is similar in some respects to Moggi's translation between meta-languages, in that the language described is translated to an explicitly-typed "internal language" IL, and the treatment of types is stressed. (The translation is expressed using inference rules rather than LF, but that difference does not seem to be so crucial.) The main differences from monadic semantics are that the internal language is itself defined *operationally*, using evaluation contexts, rather than denotationally (although Harper and Stone conjecture that "it would also be feasible to give it a domain-theoretic interpretation as in denotational semantics"); also, it does not involve any explicit monadic structure.

It seems that type-theoretic interpretation is also quite similar in nature to action semantics, with the internal language playing the rôle of action notation. Here, apart from the disregard for types in action semantics, the main difference is that the internal language used in [6] is specifically intended for representing the semantics of ML constructs, whereas action notation aims to be unbiased towards any particular class of languages. Harper and Stone's conjecture that "languages such as Caml, Haskell, and Scheme could be interpreted into an internal language substantially similar to the one given" may well be true, but it remains doubtful whether it would be convenient to use the same internal language for Java, for instance—or even for the forthcoming ML2000. A more superficial difference is the use of inference rules, rather than the semantic equations used in action semantics, to specify the translation to the internal language.

To assess the modularity of type-theoretic interpretation, let us again consider the description of a pure functional language and its extensions. The translation of if-constructs into the internal language IL is described as follows [6, App. D.1]:

$$\begin{array}{l}\mathtt{if}\ exp_1\ \mathtt{then}\ exp_2\ \mathtt{else}\ exp_3 \mapsto \\ \quad \mathtt{case}^{\mathtt{Bool}}\ exp_1\ \mathtt{of}\ \lambda var : \mathtt{Bool}_{\overline{\mathtt{false}}}.exp_3, \lambda var : \mathtt{Bool}_{\overline{\mathtt{true}}}.exp_2\ \mathtt{end}\end{array} \tag{76}$$

(In fact the if-expression above is regarded as a "derived form" in IL, available for use also in IL terms occurring as the result of translation.) IL has equality on numbers, so no translation at all is needed for $exp_1 \mathtt{=} exp_2$.

Since references and exceptions are directly supported by IL, it is unsurprising that when the described language is extended with the corresponding bench-mark constructs, no reformulation of the translation of the above functional constructs is required. A dereferencing expression `deref` *exp* is translated to an application of the IL constant `get`, and an assignment exp_1 `:=` exp_2 is translated to an application of the constant `set`. The exception `fail` is translated to a corresponding constant. The translation of catch-constructs into the internal language IL is described as follows [6, App. D.1]:

$$\begin{array}{l} \texttt{catch}\ exp_1\ \texttt{with}\ exp_2 \mapsto \\ \ \texttt{handle}\ exp_1\ \texttt{with}\ (\lambda var : \texttt{Tagged}. \\ \ \ \texttt{iftagof}\ var\ \texttt{is}\ basis.\mathrm{fail}^*.\mathrm{tag} \\ \ \ \ \texttt{then}\ \lambda var : \texttt{Unit}.exp_2\ \texttt{else}\ \texttt{raise}^{con}\, var) \end{array} \tag{77}$$

However, the internal language provides no support at all for concurrency (or even for nondeterminism), so it seems that any interpretation of concurrency constructs would be quite indirect.

5 Conclusion

It would be unwise to try to draw any definite conclusions regarding modularity in meta-languages from the simple bench-mark adopted here: perhaps the bench-mark should have included continuations, or excluded concurrency, or maybe it is biased towards the author's favourite meta-languages? Nevertheless, it is hoped that this survey has cast some light on the relative strengths and weaknesses of the considered frameworks, and may perhaps inspire the development of new meta-languages with even better modularity.

One major omission from this paper (due to shortage of time rather than lack of interest) is an assessment of the Abstract State Machine (ASM) operational framework, which has been used to give "a programmer-friendly modular definition of the semantics of Java" [2].

Acknowledgments Thanks to the anonymous referees for useful suggestions and comments. This paper was written while on leave from BRICS, visiting Centre of Informatics, UFPE, Recife, Brazil, supported by a grant from CNPq. BRICS (Centre for Basic Research in Computer Science) was established by the Danish National Research Foundation, in collaboration with the Universities of Aarhus and Aalborg, Denmark.

References

1. 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.

2. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer-Verlag, 1998. `http://www.eecs.umich.edu/gasm/`.
3. 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.
4. M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 193–217. North-Holland, 1987.
5. W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisumulation for core CML. *J. Functional Programming*, 8(5):447–451, 1998.
6. 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.
7. 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.
8. S. B. Lassen, P. D. Mosses, and D. A. Watt. A proposal for a revised Action Notation (introduction). In P. D. Mosses and H. Moura, editors, *Proc. 3rd Intl. Workshop on Action Semantics, Recife, Brazil*, Notes Series. BRICS, Dept. of Computer Science, Univ. of Aarhus, 2000. To appear.
9. 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.
10. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1997.
11. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.
12. E. Moggi. Metalanguages and applications. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute. CUP, 1997. `http://www.disi.unige.it/person/MoggiE/ftp/ML-notes.dvi.gz`.
13. P. D. Mosses. Making denotational semantics less concrete. In *Proc. Int. Workshop on Semantics of Programming Languages, Bad Honnef*, pages 102–109. Abteilung Informatik, Universität Dortmund, 1977. Bericht nr. 41.
14. P. D. Mosses. Abstract semantic algebras! In *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*, pages 45–71. North-Holland, 1983.
15. P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
16. P. D. Mosses. A practical introduction to denotational semantics. In *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 1–49. Springer-Verlag, 1991.
17. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
18. 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.
19. 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 [20].

20. 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 [19].
21. 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`.
22. P. D. Mosses and M. A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *LNCS*, pages 461–479. Springer-Verlag, 1994.
23. P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.
24. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, Chichester, UK, 1992.
25. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
26. J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.
27. J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Computer Science Dept., Cornell Univ., 1992. Tech. Rep. TR 92-1285.
28. D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *Proc. LICS'97.* IEEE, 1997.
29. K. Wansbrough. A modular monadic action semantics. Master's thesis, Dept. of Computer Science, Univ. of Auckland, Feb. 1997.
30. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997. `http://www.cl.cam.ac.uk/users/kw217/research/`.
31. D. A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, 1991.
32. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Preliminary version in Rice TR 91-160.

Recent BRICS Report Series Publications

RS-00-50 Peter D. Mosses. *Modularity in Meta-Languages*. December 2000. 19 pp. Appears in *2nd Workshop on Logical Frameworks and Meta-Languages*, LFM '00 Proceedings, 2000.

RS-00-49 Ulrich Kohlenbach. *Higher Order Reverse Mathematics*. December 2000. 18 pp.

RS-00-48 Marcin Jurdziński and Jens Vöge. *A Discrete Stratety Improvement Algorithm for Solving Parity Games*. December 2000.

RS-00-47 Lasse R. Nielsen. *A Denotational Investigation of Defunctionalization*. December 2000. Presented at *16th Workshop on the Mathematical Foundations of Programming Semantics*, MFPS '00 (Hoboken, New Jersey, USA, April 13–16, 2000).

RS-00-46 Zhe Yang. *Reasoning About Code-Generation in Two-Level Languages*. December 2000.

RS-00-45 Ivan B. Damgård and Mads J. Jurik. *A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System*. December 2000. 18 pp. Appears in Kim, editor, *Fourth International Workshop on Practice and Theory in Public Key Cryptography*, PKC '01 Proceedings, LNCS 1992, 2001, pages 119–136. This revised and extended report supersedes the earlier BRICS report RS-00-5.

RS-00-44 Bernd Grobauer and Zhe Yang. *The Second Futamura Projection for Type-Directed Partial Evaluation*. December 2000. To appear in *Higher-Order and Symbolic Computation*. This revised and extended report supersedes the earlier BRICS report RS-99-40 which in turn was an extended version of Lawall, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '00 Proceedings, 2000, pages 22–32.

RS-00-43 Claus Brabrand, Anders Møller, Mikkel Christensen, Ricky, and Michael I. Schwartzbach. *PowerForms: Declarative Client-Side Form Field Validation*. December 2000. 21 pp. To appear in *World Wide Web Journal*, 4(3), 2000.

RS-00-42 Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. *The `<bigwig>` Project*. December 2000. 25 pp.