Basic Research in Computer Science

Semantics, Modularity, and Rewriting Logic

Peter D. Mosses

BRICS Report Series **RS-98-42**

ISSN 0909-0878 **December 1998**

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/98/42/`

Semantics, Modularity, and Rewriting Logic

Peter D. Mosses*

BRICS,†Dept. of Computer Science, University of Aarhus
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark

Abstract

A complete formal semantic description of a practical programming language (such as Java) is likely to be a lengthy document, regardless of which semantic framework is being used. Good modularity of the description is important to the person(s) developing it, to facilitate reuse, change, and extension. Unfortunately, the conventional versions of the major semantic frameworks have rather poor modularity.

In this paper, we first recall some approaches that improve the modularity of denotational semantics, namely *action semantics*, *modular monadic semantics*, and a hybrid framework that combines these: *modular monadic action semantics*. We then address the issue of modularity in *operational semantics*, which appears to have received comparatively little attention so far, and report on some preliminary investigations of how one might achieve the same kind of modularity in structural operational semantics as the use of monad transformers can provide in denotational semantics—this is the main technical contribution of the paper. Finally, we briefly consider the representation of structural operational semantics in rewriting logic, and speculate on the possibility of using it to interpret programs in the described language. Providing powerful meta-tools for such semantics-based interpretation is an interesting potential application of rewriting logic; good modularity of the semantic descriptions may be crucial for the practicality of using the tools.

Much of the paper consists of (very) simple examples of semantic descriptions in the various frameworks, illustrating the degree of reformulation needed when extending the described language—a strong

*E-mail: pdmosses@brics.dk

†Basic Research in Computer Science, Centre of the Danish National Research Foundation

indicator of modularity. Throughout, it is assumed that the reader has some familiarity with the concepts and notation of denotational and structural operational semantics. Familiarity with the basic notions of monads and monad transformers is not a prerequisite.

1 Modularity in Denotational Semantics

It is well-known [5, 12, 13, 22] that the cause of poor modularity in conventional denotational semantic descriptions is the unrestricted use of (typed) λ-notation to specify semantic entities. When the described language is extended with unanticipated new constructs, the domains of denotations may need to be changed, and then the description of the old constructs may have to be completely reformulated to adapt it to the new domains. A small-scale illustration of the poor extensibility in denotational semantics is provided in Appendix A.

Action semantics [15, 16, 18, 23] improves the modularity of denotational semantics by taking denotations to be so-called *actions*, which are expressed using a fixed *action notation* consisting of various primitives and combinators—a few of them are listed in Appendix B. The primary interpretation of action notation is, in contrast to that of λ-notation, operational, and it is defined [15] using structural operational semantics (and a derived testing equivalence). Action notation provides direct support for specifying control flow, data flow, scopes of bindings, side-effects, procedural abstraction, and (asynchronous) communication between concurrent processes. The high degree of extensibility obtained in action semantics is illustrated in Appendix C; larger-scale illustrations have been given elsewhere [15, 17].

Also the use of *monads* in denotational semantic descriptions can improve their modularity [12]. Briefly, a monad distinguishes between *values* and *computations*, and provides a (polymorphic) operator, here written as infix >>=, for composing computations, as well as one, written *return*, that turns a value into a computation which simply computes that value. The use of this composition operator is independent of how computations are represented—in marked contrast to ordinary function composition.

Complex monads can generally be built systematically by composing a series of *monad transformers*, each of which may provide various operators (apart from composition and returning a value). A systematic approach to lifting operators through monad transformers has been developed in the framework called *modular monadic semantics* [5]; its implementation in the higher-order functional programming language Gofer provides modu-

lar semantics-based interpreters. The high degree of extensibility obtained in modular monadic semantics is illustrated in Appendix D. This framework appears to be simpler to use than other approaches to lifting monad transformers [3, 20], even though (or perhaps because?) the latter are based more directly on sophisticated category-theoretical foundations.

The framework of *modular monadic action semantics* [21, 22] combines the two approaches of modular monadic semantics and action semantics by using the former to give a modular denotational definition of action notation—replacing its original structural operational semantics, which has rather poor modularity (see the next section). This approach facilitates the extension of action notation to support constructs that it currently lacks, such as first-class continuations. (A disciplined—but not explicitly monadic—use of action combinators in denotational descriptions has also been proposed by the present author [14].) However, the part of action notation concerned with communication and concurrency has been omitted, and may be difficult to incorporate: a proper semantic treatment would seem to require non-(ω-)continuous functions on power domains to model fairness. Another drawback is that the operational consequences of the monadic semantics for action notation may be excessively difficult to grasp for those not already well-versed in the usual techniques for encoding computations as higher-order functions.

Modular monadic action semantics was motivated by the lack of modularity of the original structural operational semantics of action notation. Is that description is *inherently* non-modular? or could it perhaps be reformulated so as to have an acceptable degree of modularity and extensibility? Could one achieve good modularity, comparable to that obtained by the use of action notation or monads in denotational semantics, generally in structural operational semantics?

2 Modularity in Structural Operational Semantics

In his Aarhus lecture notes [19], where he first proposed structural operational semantics (SOS) as a general framework for describing programming languages, Gordon Plotkin wrote:

> *As regards modularity we just hope that if we get the other things in a reasonable shape, the current ideas for imposing modularity on specifications will prove useful.*

Unfortunately, that hope appears to have been in vain: conventional SOS descriptions have just as poor modularity as conventional denotational descriptions, since the semantic components of the transition relation (environments, stores, etc.) are made explicit in every rule, and a complete reformulation is needed when adding further components. A simple illustration of the problem is provided in Appendix E.

The problem of poor modularity seems not to arise when using SOS to give descriptions of process algebras—the huge success of this application of SOS may explain why modularity has not been as much as concern here as in denotational semantics. Incidentally, so-called *natural semantics* [4] seems to suffer just as much from poor modularity as SOS does.

In denotational semantics, good modularity has been obtained *simply by adopting a more disciplined notation*, avoiding any reference to irrelevant semantic components when defining the semantics of each construct. The author has recently tried to do something similar for structural operational semantics; the approach is illustrated below.

Its main features are as follows:

- Whereas the values computed by program constructs are added directly to the abstract syntax of the language being described (and regarded as terminal configurations, following Plotkin [19]), all the other semantic arguments of the transition relation (e.g., environments, current and subsequent stores) are *hidden* in abstract transitions.

  By this means, the transition relation is *always* ternary, taking as arguments: (i) the current syntax; (ii) the semantic "action" of the transition being made; and (iii) the subsequent syntax or computed value.

  A special case of the "actions" here could be simply input and output labels on transitions, as needed when using SOS for process algebra; but in general, the algebra of actions includes sequencing:

- Sequencing of semantic actions $a_1; a_2$ is associative, and silent actions τ are units.

  Sequencing of actions is usually partial, since a_1 may be followed by a_2 only if the information left by a_1 is consistent with that from which a_2 starts. Thus $a; \tau = a$ only when $a; \tau$ is defined, and similarly for $\tau; a = a$. N.B. τ is regarded as a variable, not a constant: there is a whole family of silent actions (one for each semantic state).

  The introduction of fixed notation for sequencing actions corresponds *roughly* to the introduction of monads in denotational semantics; the

silent actions distinguish transitions that are inherently unobservable, not affecting the semantic components of the current state but allowing gradual propagation of the flow of control.

- The transitive and reflexive-transitive closures of the transition relation are always available.

  These closures correspond closely to the notion of an entire computation in a monad; they are easily definable in terms of single transitions. Plotkin used them for letting a (terminating) sequence of transitions for expression evaluation give rise to a single transition for an enclosing statement, and for giving particularly simple descriptions of iterative constructs.

  Natural semantics [4] may be obtained when the only relation for which rules are given is the reflexive-transitive closure (the second "syntactic" argument of which is always a computed value). By using different notation for the single transition relation and its closure, it should be possible to integrate natural and structural operational semantics (see also [1]).

Adoption of the above discipline for SOS appears to provide the desired degree of modularity—at least for the kind of examples given in Plotkin's notes. It remains to be seen whether this approach can be applied to provide a modular operational semantics of the full action notation, and whether it can also be used to describe further constructs, such as first-class continuations. It should also be compared to other approaches [2, 8] which obtain some degree of modularity in a rather different way to that explored here, using so-called evaluation contexts.

Let us now illustrate the extensibility obtained, taking the same example language extension as for the other approaches considered above.

2.1 Initial Description

The abstract syntax of the initial language is given in Appendix A.1.1. By adding computed values to each syntactic category we obtain the configurations, exactly as in Appendix E.1.2.

For our transition relations, let us take, for each configuration set $\mathbf{X}$ whose elements evaluate to semantic values in $\mathbf{V} \subseteq \mathbf{X}$, a ternary (single-step) transition relation involving semantic action arguments from $A_{\mathbf{X}}$:

$\_ \overset{\_}{\rightarrow} \_ : \mathbf{X} \times A_{\mathbf{X}} \times \mathbf{X}$

and its reflexive-transitive closure:

$\_ \overset{\_}{\rightarrow}{}^{*} \_ : \mathbf{X} \times A_{\mathbf{X}} \times \mathbf{V}$

the latter being specified in terms of the former by:

$$\frac{X \overset{a}{\rightarrow} X' \qquad X' \overset{a'}{\rightarrow}{}^{*} v \qquad a; a' = a''}{X \overset{a''}{\rightarrow}{}^{*} v} \qquad\qquad \frac{}{v \overset{\tau}{\rightarrow}{}^{*} v}$$

Our example language includes declarations (of constants), so we shall have to include the current environment somehow. For expressions, the computed values do not include environments, and the discipline of our restriction to ternary transition relations *forces* us to take the current environment as a component of the second argument of the relation; here, it turns out to be the only component needed.

For declarations, the computed values are themselves environments—but just small ones, reflecting the bindings produced by the declarations, and not including the current environment; so again we let the current environment be the only component of the second argument of the transition relation.

Thus we take sets $A_{\mathbf{Exp}} = A_{\mathbf{Dcl}} = \mathbf{Env}$ of semantic actions. We have now to define sequencing on them. The appropriate definition appears to be to let $a; a' = \rho$ when $a = a' = \rho$, otherwise undefined. In this simple example, all the actions are essentially silent—changes to the current environment are not directly observable.

We shall need two further bits of notation, associated with environments and not depending on the language being defined. The first is an extra configuration $\mathit{lookup}(I)$ for requesting the value currently bound to a particular identifier; its only transition is specified by:

$$\frac{\rho(I) = v}{\mathit{lookup}(I) \overset{\rho}{\rightarrow} v}$$

The second overlays an environment ρ' on top of the current environment component ρ of an action a, which is here simply a itself:

$\mathit{overlay}(a, \rho') = a[\rho']$,

using the same notation for combining environments as in the other approaches illustrated in the appendices.

This completes our preparations—now for specifying the the transition rules. Note that "side-conditions" on rules are here written as premisses, for notational convenience, as in Appendix E.

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1 \ O \ E_2 \xrightarrow{a} E_1' \ O \ E_2} \qquad \frac{E_2 \xrightarrow{a} E_2'}{e_1 \ O \ E_2 \xrightarrow{a} e_1 \ O \ E_2'} \qquad \frac{e_1 \ O \ e_2 = e}{e_1 \ O \ e_2 \xrightarrow{\tau} e}$$

$$\frac{lookup(I) \xrightarrow{\tau} e}{I \xrightarrow{\tau} e} \qquad \frac{D \xrightarrow{a} D'}{\textbf{let } D \textbf{ in } E \xrightarrow{a} \textbf{let } D' \textbf{ in } E} \qquad \frac{overlay(a, \rho) = a' \quad E \xrightarrow{a'} E'}{\textbf{let } \rho \textbf{ in } E \xrightarrow{a} \textbf{let } \rho \textbf{ in } E'}$$

$$\frac{}{\textbf{let } \rho' \textbf{ in } e \xrightarrow{\tau} e} \qquad \frac{E \xrightarrow{a} E'}{\textbf{const } I = E \xrightarrow{a} \textbf{const } I = E'} \qquad \frac{}{\textbf{const } I = e \xrightarrow{\tau} (I \mapsto e)}$$

$$\frac{D_1 \xrightarrow{a} D_1'}{D_1 \textbf{ and } D_2 \xrightarrow{a} D_1' \textbf{ and } D_2} \qquad \frac{D_2 \xrightarrow{a} D_2'}{\rho_1 \textbf{ and } D_2 \xrightarrow{a} \rho_1 \textbf{ and } D_2'}$$

$$\frac{dom(\rho_1) \cap dom(\rho_2) = \emptyset}{\rho_1 \textbf{ and } \rho_2 \xrightarrow{\tau} \rho_1 \cup \rho_2}$$

2.2 A Simple Extension

The abstract syntax of the extended language is given in Appendix A.2.1. By adding computed values to each syntactic category we obtain the configurations, exactly as in Appendix E.2.2.

To cater for variable declarations and assignment commands, we need to add components to our semantic actions for holding both the current store and the subsequent store. For simplicity of notation, let us take uniform sets $A_{\mathbf{Exp}} = A_{\mathbf{Dcl}} = A_{\mathbf{Cmd}} = \mathbf{Env} \times \mathbf{S} \times \mathbf{S}$ of semantic actions (in fact the second store component for expressions could just as well be eliminated).

We have now to define sequencing on them. The appropriate definition is now to let $a_1; a_2 = (\rho, \sigma, \sigma'')$ when $a_1 = (\rho, \sigma, \sigma')$ and $a_2 = (\rho, \sigma', \sigma'')$, otherwise undefined. The silent actions are those of the form (ρ, σ, σ).

We shall need to redefine the two bits of notation associated with environments:

$$\frac{\rho(I) = v}{lookup(I) \xrightarrow{(\rho,\sigma,\sigma)} v} \qquad overlay((\rho, \sigma, \sigma'), \rho') = (\rho[\rho'], \sigma, \sigma')$$

These changes correspond roughly to the lifting of operators through monad transformers that is provided in modular monadic semantics.

Finally, we introduce three new configurations, concerned entirely with stores: *new*, for allocating a new location; $update(l, s)$, for overwriting the contents of the store at location l with value s; and $contents(l)$, for inspecting the value stored at location l. Their transitions are specified as follows:

$$\frac{(l \notin dom(\sigma) \quad \sigma[l \mapsto \bot] = \sigma'}{new \xrightarrow{(\rho,\sigma,\sigma')} l} \qquad \frac{l \in dom(\sigma) \quad \sigma[l \mapsto s] = \sigma'}{update(l,s) \xrightarrow{(\rho,\sigma,\sigma')} ()}$$

$$\frac{contents(l) \xrightarrow{(\rho,\sigma,\sigma')} s}{\sigma(l) = s \quad \sigma = \sigma'}$$

Now, *no changes to the original rules are needed at all*—one simply adds the following new rules:

$$\frac{new \xrightarrow{a} l}{\mathbf{var}\ I \xrightarrow{a} (I \mapsto l)} \qquad \frac{lookup(I) \xrightarrow{\tau} l}{I := E \xrightarrow{\tau} l := E} \qquad \frac{E \xrightarrow{a} E'}{l := E \xrightarrow{a} l := E'} \qquad \frac{update(l,s) \xrightarrow{a} ()}{l := s \xrightarrow{a} ()}$$

$$\frac{lookup(I) \xrightarrow{\tau} l \quad contents(l) \xrightarrow{\tau} s}{I \xrightarrow{\tau} s} \qquad \frac{C_1 \xrightarrow{a} C_1'}{C_1; C_2 \xrightarrow{a} C_1'; C_2} \qquad \frac{}{(); C_2 \xrightarrow{\tau} C_2}$$

3 Prototyping Semantics using Rewriting Logic

Since SOS can be represented straightforwardly in rewriting logic [7], one may in principle use systems such as Maude and ELAN (see the other papers in this volume for current references) for interpreting programs according to their specified semantics. Perhaps also Maude's object-oriented modules could be exploited to express the intended operational semantics more concisely, following the specification style for concurrent object-oriented systems illustrated in [6, 9, 10]. But an SOS description of a practical programming language is likely to be rather large—regardless of its degree of modularity—and this might provoke problems with the efficiency of the interpretation.

An alternative approach would be to exploit action semantics as an intermediate step. The action semantics of a practical programming language may itself be quite large; but the map that it specifies from programs to actions is purely functional, and can be implemented efficiently, e.g. by term rewriting. It remains to interpret actions according to the SOS of action notation, using the representation of SOS in rewriting logic. The primary advantage of this two-stage approach is that the SOS of action notation may be (at least) an order of magnitude smaller than the SOS of the programming language. The size of the actions would be somewhat larger (by a constant factor) than the corresponding programs, but at least for prototyping purposes, this expansion of the term to be interpreted should not be a source of undue inefficiency.

These proposals are currently still quite speculative, and need much further investigation and experimentation. Potentially, they could lead to a significant application of rewriting logic, providing useful meta-tools for

semantics-based interpretation of programs. Comments and suggestions concerning the approach outlined above are especially welcome at this early stage of the work.

4 Conclusion

We have considered some aspects of the connections between semantics, modularity, and rewriting logic. We have found that it seems possible to make a considerable improvement to the modularity of structural operational semantics by insisting on a disciplined use of notation for transitions, making the transition rules independent of the presence or absence of particular semantic components of the transition relations. The development of such a modular form of operational semantics is a contribution to a recently-started project at SRI International and Stanford University, which aims to exploit rewriting logic in providing useful meta-tools for logics and programming languages.

Acknowledgement

The author is grateful to the organizers of WRLA'98 for the invitation to give a talk on applications of rewriting logic to semantics, and for funding to attend the workshop; also to José Meseguer and Carolyn Talcott for the invitation to participate in their meta-tools project. After reading a draft version of this paper, Andrzej Filinski, John Hamer, José Meseguer, and David Watt provided valuable suggestions for improving not only the presentation, but also some technical details concerning the proposed approach to modular operational semantics; any remaining defects are, of course, entirely the author's own responsibility.

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

References

[1] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 51–136. Springer-Verlag, 1991.

[2] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In M. Hagiya and J. C. Mitchell, editors, *Symposium on Theoretical Aspects of Computer Software*, number 789 in LNCS, pages 244–272, Sendai, Japan, Apr. 1994. Springer-Verlag.

[3] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *Proceedings of the Conference on Category Theory and Computer Science*, Amsterdam, The Netherlands, Sept. 1993. CWI Tech. Report.

[4] G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*. Springer-Verlag, 1987.

[5] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *LNCS*, pages 219–234. Springer-Verlag, 1996.

[6] N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhoefer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*. Kluwer Academic Publishers, 1998. To appear. Also Technical Report SRI-CSL-94-07, SRI International, April 1994.

[7] 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. To appear. Also Technical Report SRI-CSL-93-05, SRI International, August 1993.

[8] I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[9] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[10] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. M. Nierstrasz, editor, *Proc. ECOOP'93, 7th European Conf., Kaiserslautern, Germany*, volume 707 of *LNCS*, pages 220–246. Springer-Verlag, 1993.

[11] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[12] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[13] P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.

[20] J. Power. Modularity in denotational semantics. In *Proc. MFPS XIII*, volume 5 of *ENTCS*, 1997. http://www.elsevier.nl/locate/entcs/volume5.html.

[21] K. Wansbrough. A modular monadic action semantics. Master's thesis, Dept. of Computer Science, Univ. of Auckland, Feb. 1997. WWW[1].

[1] http://www.dcs.gla.ac.uk/~keithw/research/msc/thesis

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

[23] D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

Appendices

For the sake of brevity, some tedious notational details are omitted below. Moreover, the possibility of program errors is ignored.

A Conventional Denotational Semantics

A.1 Initial Description

A.1.1 Abstract Syntax

(**Exp**) $E ::= N \mid E_1\ O\ E_2 \mid I \mid \textbf{let}\ D\ \textbf{in}\ E$
(**Nml**) $N ::= \ldots$
(**Opr**) $O ::= + \mid - \mid * \mid = \mid <$
(**Dcl**) $D ::= \textbf{const}\ I = E \mid D_1\ \textbf{and}\ D_2$
(**Ide**) $I ::= \ldots$

A.1.2 Domains

$e \in \mathbf{EV} = \mathbf{N} + \mathbf{B}$ (expressible values)
$n \in \mathbf{N} = \ldots$ (natural numbers)
$b \in \mathbf{B} = \ldots$ (Boolean truth-values)
$\rho \in \mathbf{Env} = \mathbf{Ide} \to \mathbf{DV}$ (environments)
$d \in \mathbf{DV} = \mathbf{N}$ (denotable values)

A.1.3 Semantic Functions

$\mathcal{E} : \mathbf{Exp} \to (\mathbf{Env} \to \mathbf{EV})$
$\mathcal{N} : \mathbf{Nml} \to \mathbf{N}$
$\mathcal{O} : \mathbf{Opr} \to (\mathbf{EV} \times \mathbf{EV} \to \mathbf{EV})$
$\mathcal{D} : \mathbf{Dcl} \to (\mathbf{Env} \to \mathbf{Env})$

$$
\begin{array}{ll}
\mathcal{E}[\![N]\!] & = \lambda\rho.\mathcal{N}[\![N]\!] \\
\mathcal{E}[\![E_1\ O\ E_2]\!] & = \lambda\rho.\mathcal{O}[\![O]\!](\mathcal{E}[\![E_1]\!]\rho, \mathcal{E}[\![E_2]\!]\rho) \\
\mathcal{E}[\![I]\!] & = \lambda\rho.\rho(I) \\
\mathcal{E}[\![\textbf{let } D \textbf{ in } E]\!] & = \lambda\rho.\mathcal{E}[E][\![(\rho[\mathcal{D}[\![D]\!]\rho])
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{D}[\![\textbf{const } I = E]\!] & = \lambda\rho.(I \mapsto \mathcal{E}[\![E]\!]\rho) \\
\mathcal{D}[\![D_1 \textbf{ and } D_2]\!] & = \lambda\rho.\mathcal{D}[\![D_1]\!]\rho \cup \mathcal{D}[\![D_2]\!]\rho
\end{array}
$$

$\ldots$ (definitions of $\mathcal{N}, \mathcal{O}$ omitted)

A.2 A Simple Extension

A.2.1 Abstract Syntax

(**Dcl**) $D ::= \ldots \mid \textbf{var } I$
(**Cmd**) $C ::= I := E \mid C_1; C_2$

A.2.2 Domains

$$
\begin{array}{lll}
d \in \mathbf{DV} = \ldots + \mathbf{Loc} & & \text{(denotable values)} \\
\sigma \in \mathbf{S} = \mathbf{Loc} \to \mathbf{SV} & & \text{(stores)} \\
s \in \mathbf{SV} = \mathbf{N} & & \text{(storable values)} \\
l \in \mathbf{Loc} = \mathbf{N} & & \text{(locations)}
\end{array}
$$

A.2.3 Auxiliary Functions

$new : \mathbf{S} \to \mathbf{Loc} \times \mathbf{S}$ (allocation)

A.2.4 Semantic Functions

The previous definitions of $\mathcal{E}, \mathcal{D}$ are no longer well-formed—the changes that have to be made in them are underlined below.

$$
\begin{array}{ll}
\mathcal{E} : \mathbf{Exp} \to & (\mathbf{Env} \to (\underline{\mathbf{S} \to \mathbf{EV}})) \\
\mathcal{D} : \mathbf{Dcl} \to & (\mathbf{Env} \to (\underline{\mathbf{S} \to \mathbf{Env} \times \mathbf{S}})) \\
\mathcal{C} : \mathbf{Cmd} \to & (\mathbf{Env} \to (\underline{\mathbf{S} \to \mathbf{S}}))
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{E}[\![N]\!] & = \lambda\rho.\lambda\sigma.\mathcal{N}[\![N]\!] \\
\mathcal{E}[\![E_1\ O\ E_2]\!] & = \lambda\rho.\overline{\lambda\sigma.}\mathcal{O}[\![O]\!](\mathcal{E}[\![E_1]\!]\rho\sigma, \mathcal{E}[\![E_2]\!]\rho\underline{\sigma}) \\
\mathcal{E}[\![I]\!] & = \lambda\rho.\overline{\lambda\sigma.}[\lambda e.e, \lambda l.\sigma(l)]\overline{(}\rho(I)) \\
\mathcal{E}[\![\textbf{let } D \textbf{ in } E]\!] & = \lambda\rho.\underline{\overline{\lambda\sigma.}(\lambda(\rho', \sigma').\mathcal{E}[\![E]\!](\rho[\rho'])\sigma')}(\mathcal{D}[\![D]\!]\rho\underline{\sigma}) \\
\mathcal{D}[\![\textbf{const } I = E]\!] & = \lambda\rho.\overline{\lambda\sigma.}((I \mapsto \mathcal{E}[\![E]\!]\rho\sigma), \underline{\sigma}) \\
\mathcal{D}[\![D_1 \textbf{ and } D_2]\!] & = \lambda\rho.\overline{\lambda\sigma.}(\lambda(\rho_1, \sigma_1).\underline{(\lambda\overline{(}\rho_2, \sigma_2).(\rho_1 \cup \rho_2, \sigma_2))}(\mathcal{D}[\![D_2]\!]\rho\underline{\sigma_1}))\\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathcal{D}[\![D_1]\!]\rho\sigma) \\
\mathcal{D}[\![\textbf{var } I]\!] & = \lambda\rho.\lambda\sigma.(\lambda(l, \sigma').\overline{(}I \mapsto l, \sigma'))(new(\sigma)) \\
\mathcal{C}[\![I := E]\!] & = \lambda\rho.\lambda\sigma.\sigma[\rho(I) \mapsto \mathcal{E}[\![E]\!]\rho\sigma] \\
\mathcal{C}[\![C_1; C_2]\!] & = \lambda\rho.\lambda\sigma.\mathcal{C}[\![C_2]\!]\rho(\mathcal{C}[\![C_1]\!]\rho\sigma)
\end{array}
$$

B Action Notation

Here, the symbols of action notation that are used in Appendix C are merely listed. Explanations of their intended operational interpretation may be found in [15, 16] (although the reader may well be able to guess it from the words used, and from the examples in Appendix C, which are for the same language constructs as in Appendix A.)

Action combinators: (arguments and results of sort *action*)
*_and_* , *_then_* , *_and then_* , *_or_* , *furthermore_* , *_hence_* , . . .

Action primitives: (arguments of sort *yielder*, results of sort *action*)
*give_* , *bind_to_* , *store_in_* , *allocate_* , . . .

Yielders: (arguments of sort *yielder*)
*given_* , *given_#_* , *it*, *the_bound to_* , *the_stored in_* , . . .

Data sorts: (subsorts of *yielder*)
datum, *bindable*, *storable*, *cell*, *truth*, . . .

C Action Semantics

C.1 Initial Description

C.1.1 Abstract Syntax

See Appendix A.1.1.

C.1.2 Semantic Entities

includes: *Action Notation* [15].
introduces: *value*, *number*.
value $= number \mid truth$.
bindable $= number$.
datum $\geq value \mid bindable$.

C.1.3 Semantic Functions

includes: *Abstract Syntax*, *Semantic Entities*.
introduces: *evaluate_* , *value of_* , *result of_* , *elaborate_* .
*evaluate_* : **Exp** $\rightarrow$ *action*[*giving a value*].
*value of_* : **Nml** $\rightarrow$ *number*.
*result of_* : **Opr** $\rightarrow$ *yielder*[*of a value*].
*elaborate_* : **Dcl** $\rightarrow$ *action*[*binding*].

$$
\begin{array}{ll}
\mathit{evaluate}[\![N]\!] & = \mathit{give\ value\ of\ } N \\
\mathit{evaluate}[\![E_1\ O\ E_2]\!] & = (\mathit{evaluate\ } E_1 \mathit{\ and\ then\ evaluate\ } E_2) \\
 & \quad \mathit{then\ give\ result\ of\ } O \\
\mathit{evaluate}[\![I]\!] & = \mathit{give\ the\ value\ bound\ to\ } I \\
\mathit{evaluate}[\![\textbf{let } D \textbf{ in } E]\!] & = (\mathit{furthermore\ elaborate\ } D) \mathit{\ hence\ evaluate\ } E \\
\mathit{elaborate}[\![\textbf{const } I = E]\!] & = \mathit{evaluate\ } E \mathit{\ then\ bind\ } I \mathit{\ to\ the\ given\ value} \\
\mathit{elaborate}[\![D_1 \textbf{ and } D_2]\!] & = \mathit{elaborate\ } D_1 \mathit{\ and\ elaborate\ } D_2
\end{array}
$$

. . . (definitions of *valueof_*, *resultof_* omitted)

C.2 A Simple Extension

Only minor changes, underlined below, are needed to the initial description, to take account of the extension of the language with imperative features.

C.2.1 Abstract Syntax

See Appendix A.2.1.

C.2.2 Semantic Entities

$bindable = number \mid cell.$
$storable = number.$
$datum \geq value \mid bindable \mid storable \mid cell.$

C.2.3 Semantic Functions

introduces: $execute\_$.
$elaborate\_ : \mathbf{Dcl} \rightarrow action[binding \mid storing]$
$execute\_ : \mathbf{Cmd} \rightarrow action[storing]$

$evaluate[\![I]\!] = give\ the\ value\ bound\ to\ I$
$\quad or\ give\ the\ value\ stored\ in\ the\ cell\ bound\ to\ I$
$elaborate[\![\mathbf{var}\ I]\!] = allocate\ a\ cell\ then\ bind\ I\ to\ it$
$execute[\![I := E]\!] = (give\ the\ cell\ bound\ to\ I\ and\ evaluate\ E)$
$\quad then\ store\ the\ given\ value\#2\ in\ the\ given\ cell\#1$
$execute[\![C_1; C_2]\!] = execute\ C_1\ and\ then\ execute\ C_2$

D Modular Monadic Semantics

D.1 Initial Description

D.1.1 Abstract Syntax

See Appendix A.1.1.

D.1.2 Monad

See Appendix A.1.2 for the domains used below.

$return : a \rightarrow M\ a$
$\_ >>= \_ : M\ a \times (a \rightarrow M\ b) \rightarrow M\ b$

type $M\ a = EnvT\ \mathbf{Env}\ Id\ a$ (i.e., $M\ a = Env \rightarrow a$ for any type a)

$rdEnv : M\ \mathbf{Env}$ (returns the current environment)
$inEnv : \mathbf{Env} \rightarrow (M\ a \rightarrow M\ a)$ (computes in the argument environment)

D.1.3 Semantic Functions

$\mathcal{E}$: **Exp** $\to$ M **EV**
$\mathcal{N}$: **Nml** $\to$ **N**
$\mathcal{O}$: **Opr** $\to$ (**EV** $\times$ **EV** $\to$ **EV**)
$\mathcal{D}$: **Dcl** $\to$ M **Env**

$$\begin{aligned}
\mathcal{E}[\![N]\!] &= \mathit{return}(\mathcal{N}[\![N]\!]) \\
\mathcal{E}[\![E_1\ O\ E_2]\!] &= \mathcal{E}[\![E_1]\!] >>= \lambda e_1.\ \mathcal{E}[\![E_2]\!] >>= \lambda e_2.\ \mathit{return}(\mathcal{O}[\![O]\!](e_1, e_2)) \\
\mathcal{E}[\![I]\!] &= \mathit{rdEnv} >>= \lambda\rho.\ \mathit{return}(\rho(I)) \\
\mathcal{E}[\![\textbf{let } D \textbf{ in } E]\!] &= \mathit{rdEnv} >>= \lambda\rho.\ \mathcal{D}[\![D]\!] >>= \lambda\rho'.\ \mathit{inEnv}(\rho[\rho'])(\mathcal{E}[\![E]\!])
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}[\![\textbf{const } I = E]\!] &= \mathcal{E}[\![E]\!] >>= \lambda e.\ \mathit{return}(I \mapsto e) \\
\mathcal{D}[\![D_1 \textbf{ and } D_2]\!] &= \mathcal{D}[\![D_1]\!] >>= \lambda\rho_1.\ \mathcal{D}[\![D_2]\!] >>= \lambda\rho_2.\ \mathit{return}(\rho_1 \cup \rho_2)
\end{aligned}$$

... (definitions of $\mathcal{N}, \mathcal{O}$ omitted)

D.2 A Simple Extension

Only minor changes, underlined below, are needed to the initial description, to take account of the extension of the language with imperative features.

D.2.1 Abstract Syntax

See Appendix A.2.1.

D.2.2 Monad

See Appendix A.2.2 for the domains used below.

type $M\ a = \mathit{EnvT}$ **Env** $(\underline{\mathit{StateT}\ \mathbf{S}\ \mathit{Id}})\ a$
(i.e., $M\ a = \mathit{Env} \to (\underline{\mathbf{S} \to a \times \mathbf{S}})$ for any type a)

alloc : M **Loc**	(allocating storage)
read : **Loc** $\to M$ **SV**	(updating storage)
write : **Loc** $\times$ **SV** $\to M()$	(inspecting storage)

D.2.3 Semantic Functions

$\mathcal{C}$: **Cmd** $\to M()$

$$\begin{aligned}
\mathcal{E}[\![I]\!] &= \mathit{rdEnv} >>= \lambda\rho.\ \underline{[\lambda e.\ \mathit{return}(e), \lambda l.\ \mathit{read}(l)]}(\rho(I)) \\
\mathcal{D}[\![\textbf{var } I]\!] &= \mathit{alloc} >>= \lambda l.\ \mathit{return}(I \mapsto l)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\![I := E]\!] &= \mathit{rdEnv} >>= \lambda\rho.\ \mathcal{E}[\![E]\!] >>= \lambda e.\ \mathit{write}(\rho(I), e) \\
\mathcal{C}[\![C_1; C_2]\!] &= \mathcal{C}[\![C_1]\!] >>= \lambda().\ \mathcal{C}[\![C_2]\!]
\end{aligned}$$

E Structural Operational Semantics

E.1 Initial Description

E.1.1 Abstract Syntax

See Appendix A.1.1.

E.1.2 Configurations

The following productions extend the abstract syntax so that constructs may be replaced by their values (which will be terminal configurations for the corresponding transition relations below):

(**Exp**) $E ::= \ldots \mid e$
(**Nml**) $N ::= \ldots \mid n$
(**Dcl**) $D ::= \ldots \mid \rho$

For brevity and uniformity, let $e \in \mathbf{EV}$, $n \in \mathbf{N}$, $\rho \in \mathbf{Env}$, etc., range over the domains specified in Appendix A.1.2 (although here, the domains should really be regarded as ordinary sets). Note that our example language does not already contain literal constants for all expressible values: truth-values need to be added.

E.1.3 Transition Relations

$\_ \vdash \_ \to \_ : \mathbf{Env} \times \mathbf{Exp} \times \mathbf{Exp}$
$\_ \vdash \_ \to \_ : \mathbf{Env} \times \mathbf{Dcl} \times \mathbf{Dcl}$

E.1.4 Transition Rules

"Side-conditions" on rules are here written as premisses, for notational convenience.

$$\frac{\rho \vdash E_1 \to E_1'}{\rho \vdash E_1\ O\ E_2 \to E_1'\ O\ E_2} \qquad \frac{\rho \vdash E_2 \to E_2'}{\rho \vdash e_1\ O\ E_2 \to e_1\ O\ E_2'} \qquad \frac{e_1\ O\ e_2 = e}{\rho \vdash e_1\ O\ e_2 \to e}$$

$$\frac{\rho(I) = e}{\rho \vdash I \to e} \qquad \frac{\rho \vdash D \to D'}{\rho \vdash \mathbf{let}\ D\ \mathbf{in}\ E \to \mathbf{let}\ D'\ \mathbf{in}\ E} \qquad \frac{\rho[\rho'] \vdash E \to E'}{\rho \vdash \mathbf{let}\ \rho'\ \mathbf{in}\ E \to \mathbf{let}\ \rho'\ \mathbf{in}\ E'}$$

$$\frac{}{\rho \vdash \mathbf{let}\ \rho'\ \mathbf{in}\ e \to e} \qquad \frac{\rho \vdash E \to E'}{\rho \vdash \mathbf{const}\ I = E \to \mathbf{const}\ I = E'}$$

$$\frac{}{\rho \vdash \mathbf{const}\ I = d \to (I \mapsto d)} \qquad \frac{\rho \vdash D_1 \to D_1'}{\rho \vdash D_1\ \mathbf{and}\ D_2 \to D_1'\ \mathbf{and}\ D_2}$$

$$\frac{\rho \vdash D_2 \to D_2'}{\rho \vdash \rho_1 \textbf{ and } D_2 \to \rho_1 \textbf{ and } D_2'} \qquad \frac{dom(\rho_1) \cap dom(\rho_2) = \emptyset}{\rho \vdash \rho_1 \textbf{ and } \rho_2 \to \rho_1 \cup \rho_2}$$

E.2 A Simple Extension

E.2.1 Abstract Syntax

See Appendix A.2.1.

E.2.2 Configurations

(**Cmd**) $C ::= \ldots \mid ()$

The single value () represents merely the termination of a command. Note that our example language does not already contain a null or skip command, which might have been used instead of ().

E.2.3 Transition Relations

$\_ \vdash \_ \to \_ : \textbf{Env} \times (\textbf{Exp} \times \textbf{S}) \times (\textbf{Exp} \times \textbf{S})$
$\_ \vdash \_ \to \_ : \textbf{Env} \times (\textbf{Dcl} \times \underline{\textbf{S}}) \times (\textbf{Dcl} \times \underline{\textbf{S}})$
$\_ \vdash \_ \to \_ : \textbf{Cmd} \times (\textbf{Cmd} \times \underline{\textbf{S}}) \times (\textbf{Cmd} \times \underline{\textbf{S}})$

The set **S** of stores is as specified in Appendix A.2.2.

E.2.4 Transition Rules

The previously-given rules are no longer well-formed—the changes that have to be made in them are underlined below.

$$\frac{\rho \vdash \langle E_1, \underline{\sigma}\rangle \to \langle E_1', \underline{\sigma'}\rangle}{\rho \vdash \langle E_1 \ O \ E_2, \underline{\sigma}\rangle \to \langle E_1' \ O \ E_2, \underline{\sigma'}\rangle} \qquad \frac{\rho \vdash \langle E_2, \underline{\sigma}\rangle \to \langle E_2', \underline{\sigma'}\rangle}{\rho \vdash \langle e_1 \ O \ E_2, \underline{\sigma}\rangle \to \langle e_1 \ O \ E_2', \underline{\sigma'}\rangle}$$

$$\frac{e_1 \ O \ e_2 = e}{\rho \vdash \langle e_1 \ O \ e_2, \underline{\sigma}\rangle \to \langle e, \underline{\sigma}\rangle}$$

$$\frac{\rho(I) = e}{\rho \vdash \langle I, \underline{\sigma}\rangle \to \langle e, \underline{\sigma}\rangle} \qquad \frac{\rho(I) = l \quad \sigma(l) = s}{\rho \vdash \langle I, \sigma\rangle \to \langle s, \sigma\rangle}$$

$$\frac{\rho \vdash \langle D, \underline{\sigma}\rangle \to \langle D', \underline{\sigma'}\rangle}{\rho \vdash \langle \textbf{let } D \textbf{ in } E, \underline{\sigma}\rangle \to \langle \textbf{let } D' \textbf{ in } E, \underline{\sigma'}\rangle}$$

$$\frac{\rho[\rho'] \vdash \langle E, \underline{\sigma}\rangle \to \langle E', \underline{\sigma'}\rangle}{\rho \vdash \langle \textbf{let } \rho' \textbf{ in } E, \underline{\sigma}\rangle \to \langle \textbf{let } \rho' \textbf{ in } E', \underline{\sigma'}\rangle} \qquad \frac{}{\rho \vdash \langle \textbf{let } \rho' \textbf{ in } e, \underline{\sigma}\rangle \to \langle e, \underline{\sigma}\rangle}$$

$$\frac{\rho \vdash \langle E, \underline{\sigma}\rangle \to \langle E', \underline{\sigma'}\rangle}{\rho \vdash \langle \textbf{const } I = E, \underline{\sigma}\rangle \to \langle \textbf{const } I = E', \underline{\sigma'}\rangle}$$

$$\frac{}{\rho \vdash \langle \textbf{const } I = d, \underline{\sigma}\rangle \to \langle (I \mapsto d), \underline{\sigma}\rangle}$$

$$\frac{\rho \vdash \langle D_1, \underline{\sigma}\rangle \to \langle D_1', \underline{\sigma'}\rangle}{\rho \vdash \langle D_1 \textbf{ and } D_2, \underline{\sigma}\rangle \to \langle D_1' \textbf{ and } D_2, \underline{\sigma'}\rangle}$$

$$\frac{\rho \vdash \langle D_2, \underline{\sigma}\rangle \to \langle D_2', \underline{\sigma'}\rangle}{\rho \vdash \langle \rho_1 \textbf{ and } D_2, \underline{\sigma}\rangle \to \langle \rho_1 \textbf{ and } D_2', \underline{\sigma'}\rangle} \qquad \frac{dom(\rho_1) \cap dom(\rho_2) = \emptyset}{\rho \vdash \langle \rho_1 \textbf{ and } \rho_2, \underline{\sigma}\rangle \to \langle \rho_1 \cup \rho_2, \underline{\sigma}\rangle}$$

$$\frac{l \notin dom(\sigma))}{\rho \vdash \langle \textbf{var } I, \sigma\rangle \to \langle (I \mapsto l), \sigma[l \mapsto \bot]\rangle} \qquad \frac{\rho \vdash \langle E, \sigma\rangle \to \langle E', \sigma'\rangle}{\rho \vdash \langle I := E, \sigma\rangle \to \langle I := E', \sigma'\rangle}$$

$$\frac{\rho(I) = l}{\rho \vdash \langle I := s, \sigma\rangle \to \sigma[I \mapsto s]} \qquad \frac{\rho \vdash \langle C_1, \sigma\rangle \to \langle C_1', \sigma'\rangle}{\rho \vdash \langle C_1; C_2, \sigma\rangle \to \langle C_1'; C_2, \sigma'\rangle}$$

$$\frac{}{\rho \vdash \langle (); C_2, \sigma\rangle \to \langle C_2, \sigma\rangle}$$

Recent BRICS Report Series Publications

RS-98-42 **Peter D. Mosses.** ***Semantics, Modularity, and Rewriting Logic*****. December 1998. 20 pp. Appears in Kirchner and Kirchner, editors,** ***International Workshop on Rewriting Logic and its Applications,*** **WRLA '98 Proceedings, ENTCS 15, 1998.**

RS-98-41 **Ulrich Kohlenbach.** ***The Computational Strength of Extensions of Weak König's Lemma*****. December 1998. 23 pp.**

RS-98-40 **Henrik Reif Andersen, Colin Stirling, and Glynn Winskel.** ***A Compositional Proof System for the Modal μ-Calculus*****. December 1998. 29 pp.**

RS-98-39 **Daniel Fridlender.** ***An Interpretation of the Fan Theorem in Type Theory*****. December 1998. 15 pp. To appear in** ***International Workshop on Types for Proofs and Programs 1998,*** **TYPES '98 Selected Papers, LNCS, 1999.**

RS-98-38 **Daniel Fridlender and Mia Indrika.** ***An n-ary `zipWith` in Haskell*****. December 1998. 12 pp.**

RS-98-37 **Ivan B. Damgård, Joe Kilian, and Louis Salvail.** ***On the (Im)possibility of Basing Oblivious Transfer and Bit Commitment on Weakened Security Assumptions*****. December 1998. 22 pp. To appear in** ***Advances in Cryptology: International Conference on the Theory and Application of Cryptographic Techniques,*** **EUROCRYPT '99 Proceedings, LNCS, 1999.**

RS-98-36 **Ronald Cramer, Ivan B. Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin.** ***Efficient Multiparty Computations with Dishonest Minority*****. December 1998. 19 pp. To appear in** ***Advances in Cryptology: International Conference on the Theory and Application of Cryptographic Techniques,*** **EUROCRYPT '99 Proceedings, LNCS, 1999.**

RS-98-35 **Olivier Danvy and Zhe Yang.** ***An Operational Investigation of the CPS Hierarchy*****. December 1998.**

RS-98-34 **Peter G. Binderup, Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum.** ***The Complexity of Identifying Large Equivalence Classes*****. December 1998. 15 pp.**