A Modular Monadic Action Semantics

Keith Wansbrough and John Hamer
University of Auckland

A Modular Monadic Action Semantics[*]

Keith Wansbrough *John Hamer*

Department of Computer Science
University of Auckland
Auckland, New Zealand
{keith-w,j-hamer}@cs.auckland.ac.nz

Abstract

A domain-specific language (DSL) is a framework which is designed to precisely meet the needs of a particular application. Domain-specific languages exist for a variety of reasons. As productivity tools, they are used to make application prototyping and development faster and more robust in the presence of evolving requirements. Furthermore, by bridging the "semantic gap" between an application domain and program code, DSLs increase the opportunity to apply formal methods in proving properties of an application.

In this paper, we contribute a synthesis of two existing systems that address the problem of providing *sound* semantic descriptions of *realistic* programming languages: action semantics and modular monadic semantics. The resulting synthesis, modular monadic action semantics, is compatible with action semantics yet adds true modularity and allows domain specific specifications to be made at a variety of levels.

1 Introduction

"I'd rather write programs to write programs than write programs." (Programming proverb).

A domain-specific language (DSL) is a framework which is designed to precisely meet the needs of a particular application. Domain-specific languages exist for a variety of reasons. As productivity tools, they are used to make application prototyping and development faster and more robust in the presence of evolving requirements. Furthermore, by bridging the "semantic gap" between an application domain and program code, DSLs increase the opportunity to apply formal methods in proving properties of an application.

Designing and implementing a domain specific language is, however, problematic. Putting aside the requirement for an efficient implementation, the language needs to be: clearly and precisely defined; modular enough to change incrementally; and amenable to formal reasoning. Different tools exist that address these varied requirements, but few make any attempt to address them all. Compiler generators provide efficient implementations, but they typically have weak formal properties. We do not consider such systems further in this paper. Semantic formalisms, such as denotational semantics and structural operational semantics, have richly developed theories but are difficult to use and lack modularity.

The conceptual distance between the high-level language and established semantic formalisms is huge. Writing such specifications is a tedious and difficult task. Natural concepts in the high-level language must be accurately simulated by constructs built from the small range of primitives in the semantic formalism. The probability of introducing errors is large, and the maintainability of the specification is poor.

Three attempts at solving the problem of providing *sound* semantic descriptions of *realistic* programming languages are presented here. The first is *action semantics* [Mos92], a highly readable notation with formal foundations in Peter Mosses's unified algebra and Plotkin's structural operational semantics [Plo83]. This system has proven quite popular, and has been used to specify a number of existing and evolving languages.

The second is Hudak, Liang and Jones' *modular monadic semantics* [LH96]. Modular monadic semantics is a structured form of denotational semantics, embedded in the Haskell language. As a relatively new system it is yet to gain widespread use, but it has a number of highly attractive features, foremost its excellent modularity properties.

These two systems can be seen as competing for the same market (Figure 1(c) and (d)): both are attempts to reduce the conceptual distance that must be bridged when formally specifying a high-level language (compare with Figure 1(a) and (b)). Both appear to do so successfully; they are, however, different DSLs and are based in different semantic formalisms.

[*]This work was supported in part by a scholarship from the University of Auckland.

$$\begin{tikzcd}
\text{HLL} \arrow[ddd, "\text{mono.}"] & \text{HLL} \arrow[ddd, "\text{mono.}"] & \text{HLL} \arrow[dd, "\text{mod.}"] & \text{HLL} \arrow[dd, "\text{mod.}"] & \text{HLL} \arrow[d, "\text{mod.}"] \\
 & & & & \text{AS} \arrow[d, "\text{mod.}"] \\
 & & \text{AS} \arrow[d, "\text{mono.}"] & \text{MMS} \arrow[d, "\text{mod.}"] & \text{MMS} \arrow[d, "\text{mod.}"] \\
\text{SOS} & \lambda\text{-calculus} & \text{SOS} & \lambda\text{-calculus} & \text{SOS} \\
\text{(a)} & \text{(b)} & \text{(c)} & \text{(d)} & \text{(e)}
\end{tikzcd}$$

Figure 1: Formal specification techniques.

The third approach arises our observation that action semantics and modular monadic semantics have much to offer each other. Action semantics provides a friendly syntax to the language specifier, and encapsulates a substantial but fixed range of language concepts. Modular monadic semantics, on the other hand, provides a rather less friendly syntax, but with complete flexibility and extensibility afforded by its internal modularity. By combining these two systems, we obtain a tiered system containing *two* domain-specific semantic notations, featuring all the key features of action semantics along with the underlying flexibility and extensibility of modular monadic semantics (Figure 1(e)). This system we call *modular monadic action semantics* (MMAS).

The user of modular monadic action semantics specifies the high-level language using the rich set of primitives provided by action semantics. If the language requires a concept that is not present in action semantics, the specifier simply drops down one level, and uses the underlying modular monadic semantics to specify the new feature. If this is not sufficient, the modular monadic layer itself may be modified or extended as required. All this takes place in a highly modular fashion, with little or no change to existing parts of the specification. Furthermore, *theories* developed at each layer can also be developed in a modular fashion. New features can be introduced without undermining the validity of existing proofs.

In the remainder of this paper we describe action semantics, modular monadic semantics, and modular monadic action semantics in more detail, and we investigate some of the applications of MMAS. We conclude with a look at some related work, some concluding comments, and directions for future work. Full details on modular monadic action semantics can be found in Wansbrough's thesis [Wan97].

2 Action semantics

Action semantics [Mos92, Mos96a] is a notation developed over some years by Mosses at Aarhus University. As Mosses puts it,

> The primary aim of action semantics is to allow *useful* semantic descriptions of *realistic* programming languages. [Mos92, p. xv]

It aims at being simultaneously formal and readable, and achieves this goal admirably. An action semantic description is entirely formal, yet it can be intuitively understood to a surprising degree by someone with no prior knowledge of the system.

However, action semantics has two significant limitations. Firstly, it is incomplete. Not all programming language concepts can be represented directly within action semantics—only those which Mosses has chosen to build into the system. There is no provision for the extension of action semantics. Secondly, action semantics theory has not progressed very far: it is difficult to use an action semantics of a language to prove properties of that language or of programs written within it.

Despite these disadvantages, action semantics has proven quite popular and rather successful. A number of compiler generators based on action semantics exist [Ørb94, BMW92, Pal92], along with at least one tool [vDM96] for assistance in developing new action semantic descriptions. Action semantic descriptions exist for a number of real languages [Mos96a, §7.1], and others are being developed. An action semantics for an intermediate language for compilers, ANDF-FS [NT94, HT94], is currently in use in the 'real world', by the Open Software Foundation. Work is progressing in a number of directions.

In the following sections the essential features of action semantics are briefly summarised. More detail can be found in Mosses' book [Mos92], or his tutorial [Mos96b].

2.1 Structure of action semantics

An action semantic description (ASD) of a language specifies the semantics of the language by means of *actions*, representing computations. The action corresponding to a given program phrase is built up from primitive actions and action *combinators*. Data referred to by the actions may be accessed by means of *yielders*, and new types of data may be readily defined in an algebraic fashion.

Action semantics divides program behaviour into a number of *facets*, corresponding to the types of information dealt with. The *basic* facet refers to control flow. The *functional* facet refers to transient information, i.e., data passed between successive actions. The *declarative* facet refers to scoped information, i.e., bindings of tokens to data. The *imperative* facet refers to stable information, i.e., the storage of data in cells. Finally, the *communicative* facet refers to permanent information, i.e., the communication of data between distributed actions. Two further facets, the *reflective* and *directive* facets, refer to reflection and indirection. Action semantics has these seven computational concepts built in, and they can be used directly by specifications.

The primitive actions, yielders and data of action semantics are separated into these categories, and are intended to act on only one facet at a time. This automatically lends a degree of modularity to an action semantic description: actions involving only, say, basic and functional behaviour (such as expression evaluation) need not concern themselves with behaviour in other facets (such as the declarative or imperative facets). The result is a readily maintainable description: alterations to behaviour relating to one facet do not affect unrelated code (as they inevitably do when a monolithic approach is used, such as the conventional structural operational semantic or lambda-calculus approaches).

A key advantage of action semantics is its very readable notation. Action semantics uses words rather than symbols, and these are chosen in a way that allows even a reader unfamiliar with action semantics to obtain a broad impression of the intended meaning of the description. This is partially enabled by its flexible (albeit idiosyncratic) type system, which permits definition of new types by extending or specialising existing ones and definition of abbreviations for commonly-occurring patterns of notation, and behaves much more like conventional set theory than do traditional domains.

As a simple example of the use of action semantics, consider the following semantics for the 'if' statement of a conventional imperative language. In this example, execute is a semantic function which gives the semantics of statements. The brackets $[\![\cdot]\!]$ enclose the abstract syntax for the statement, and the term to the right of the equals sign is the action representing the semantics of the statement.

$$
\begin{array}{l}
\text{execute } [\![\text{ “if” } E \text{ “then” } S_1 \text{ “else” } S_2 \text{ }]\!] = \\
\quad \text{evaluate } E \text{ then} \\
\quad \left|\begin{array}{l} \mid \text{ check it and then execute } S_1 \\ \text{or} \\ \mid \text{ check not it and then execute } S_2 \text{ .} \end{array}\right.
\end{array}
$$

The action corresponding to the phrase “if” E “then” S_1 “else” S_2 begins by calling evaluate E, defined elsewhere. The result of the expression is threaded by the then combinator to the next action. This is an or, which nondeterministically executes one branch and backtracks on failure. The action check, applied to the yielder it, fails if the yielded value is false and succeeds if it is true; the other branch has the complementary test. After the test, execution passes (via and then) to an execute of the appropriate branch.

This example shows the way in which action semantics builds up actions from primitive actions, combinators, and yielders. Notice how easy the specification is to read, even without any familiarity with the notation. Also note that there is no need to thread the store through this equation, even though the evaluation of E may access or modify it. The modularity of action semantics means we need not consider facets not directly referenced.

2.2 Semantics

Action semantics is intended to be a formal notation, and in order to achieve this its own semantics must be precisely and formally defined. Mosses chose to do this by providing a low-level definition of action semantics as a *structural operational semantics* (SOS) [Plo81, Plo83], using a slight variation on Plotkin's original approach.

As an example, here are some of the equations describing the behaviour of the or combinator, used above to define the semantics of the 'if' statement. The equations are taken from [Mos92, §C.3.3.2.1].

(7) $\text{stepped } (A_1, l) \geq (A'_1\text{:Acting}, l'\text{:local-info, uncommitted}) \text{ ;}$
$[\![A_1 \; O \; A_2]\!] : [\![\text{ Intermediate “or” Intermediate }]\!] \Rightarrow$
$\text{stepped } ([\![A_1 \; O \; A_2]\!], l\text{:local-info})$
$\quad \geq (\text{simplified } [\![A'_1 \; O \; A_2]\!], l', \text{uncommitted}) \text{ .}$

(8) $\text{stepped } (A_2, l) \geq (A'_2\text{:Acting}, l'\text{:local-info, uncommitted}) \text{ ;}$
$[\![A_1 \; O \; A_2]\!] : [\![\text{ Intermediate “or” Intermediate }]\!] \Rightarrow$
$\text{stepped } ([\![A_1 \; O \; A_2]\!], l\text{:local-info})$
$\quad \geq (\text{simplified } [\![A_1 \; O \; A'_2]\!], l', \text{uncommitted}) \text{ .}$

(9) $\text{stepped } (A_1, l) \geq (A'_1\text{:Acting}, l'\text{:local-info}, c'\text{:committing}) \text{ ;}$
$[\![A_1 \; O \; A_2]\!] : [\![\text{ Intermediate “or” Intermediate }]\!] \Rightarrow$
$\text{stepped } ([\![A_1 \; O \; A_2]\!], l\text{:local-info}) \geq (A'_1, l', c') \text{ .}$

(10) $\text{stepped } (A_2, l) \geq (A'_2\text{:Acting}, l'\text{:local-info}, c'\text{:committing}) \text{ ;}$
$[\![A_1 \; O \; A_2]\!] : [\![\text{ Intermediate “or” Intermediate }]\!] \Rightarrow$
$\text{stepped } ([\![A_1 \; O \; A_2]\!], l\text{:local-info}) \geq (A'_2, l', c') \text{ .}$

Equations (7) and (8) handle uncommitted cases, and equations (9) and (10) committed cases (commitment is analagous to the action of Prolog's 'cut' operator (!): it commits to the chosen branch, prohibiting backtracking). In both cases, the equations specify nondeterministic choice of one action to be stepped. Once this is done, if the state remains uncommitted then equations (7) and (8) indicate a simplification is to be performed over the whole phrase; if the state becomes committed then equations (9) and (10) specify that the alternate branch is discarded. The result (in combination with the omitted definition of the simplify function) is a nondeterministic backtracking choice with commitment.

2.3 Problems

Action semantics provides an usable foundation on which to build specifications of programming languages. Referring back to Figure 1, it bridges the gap between high-level language and low-level formalism very effectively, providing a language that already contains the features required by the high-level language. In addition, it is highly readable and ensures a degree of modularity in specifications.

However, the features supported by action semantics are fixed: the facets of action semantics are those defined by Mosses, and no mechanism is provided to modify or extend them. The facets provided are sufficient to describe many common imperative languages; but there are language features (notably continuations, as popularised by Scheme for example) that are *not* present in action semantics. The only way to specify languages containing such features is by low-level simulation; resorting to this instantly loses all the advantages of action semantics.

One of the reasons for the fixed nature of the facets in action semantics is indicated in the diagram in Figure 1(c). While the specification of the high-level language in action semantics is done in a modular manner, Mosses' definition of action semantics in terms of structural operational semantics is done monolithically. This means that there is no real separation between facets at this level, and the defining equations for each component must correctly handle not only their own information but also the information from other, unrelated components. Without modularity we are back to the situation of Figure 1(a), where specification is tedious and error-prone, and maintainability is poor.

Another problem with action semantics lies in its theory. One of the goals of a formal specification of a language, as articulated by Mosses [Mos92, p. 4], is to use it

> ...as a basis for *reasoning* about the correctness of particular programs in relation to their specifications, and for justifying program transformations.

An action semantic description of a language certainly provides a formal specification, but is it useful? Mosses himself notes that "a decent theory for action semantics has been slow to emerge" [Mos96a, §6]. Compilers incorporating provably valid action transformations exist [Mos96a, §6.2–3]; but the closest approach to a general and tractable theory for action equivalence still covers only the basic, functional and declarative facets of action semantics—omitting the imperative and communicative facets essential to most real programming languages [Mos96a, §6.4].

3 Modular monadic semantics

Modular monadic semantics (MMS) [LHJ95, LH96] is a structured form of denotational semantics developed recently by Liang, Hudak, and Jones of Yale University based on the work of Moggi [Mog89a, Mog91a] and Espinosa [Esp93, Esp94]. It provides a very usable approach to denotational semantics with excellent flexibility and modularity properties. MMS is presented in the syntax of an existing functional language, thus permitting specifications to be directly executed.

In the following sections we very briefly note the essential features of modular monadic sematics; more details may be found in [LH96] and [LHJ95].

3.1 Structure of MMS

Modular monadic semantics specifies the semantics of a language by a mapping from terms to *computations* performed within a *monad*. The monad hides details of semantic features such as environments and stores that are used by the computation, and exposes operators allowing access to these features.

All monads have the two primitive operators $return$ and $\gg=$ (pronounced "bind"). The expression $return\ x$ represents the trivial computation with result x; the expression $c_1 \gg= \lambda v \rightarrow c_2$ represents the computation that computes c_1, binds the result to v, then computes c_2.

In addition to these essential operators, monads encapsulating semantic features provide operators to access them. For example, an environment monad might provide $rdEnv$ and $inEnv$ operators; a continuation monad would provide a $callcc$ operator.

This abstraction allows the underlying monad (representing the required semantic features) to be modified without altering the specification that uses it. Even if new features are added to the monad, the existing interface remains unchanged. Equations from one specification can be used within another, as long as the features used in the one are all found in the other.

The abstraction also allows us to ignore irrelevant details. As we manipulate the store, for example, we need not concern ourselves with preserving the environment—this is done transparently by the $return$ and $\gg=$ operators.

Alone, this is not sufficient. Even though the specification using the monad need not change as the feature set represented changes, it is clear that the monad itself must change: it must incorporate new operators, and the behaviour of existing operators (especially $return$ and $\gg=$) must change appropriately. For this reason, we introduce *monad transformers*.

A *monad transformer* is an object that transforms a monad, modifying the behaviour of its existing operators and adding new ones. In modular monadic semantics, we represent each desired semantic feature by a monad transformer, and then apply them all to a trivial monad. The resulting monad incorporates all of the features of the component monad transformers, and is used for the semantic specification of the high-level language.

The great advantage of this system its flexibility. Depending on how much support is needed, any set of monad transformers representing any set of semantic features may be combined in a modular fashion to provide that support. Monad transformers provide the power needed to encapsulate high-level semantic features, but still allow access to the low-level semantic detail.

The system is highly modular: the definition of each monad transformer is independent of the others; one need not concern oneself with handling the details of unrelated semantic features. As well as simplifying the writing of monad transformers, this also means that one may construct modular *proofs* within the system: as modular monadic semantics is simply a structured form of denotational semantics our normal proof methods still apply, but now we may deal with each semantic feature separately and rely on the system to cleanly and safely combine them.

3.2 Example

As a simple example of the use of modular monadic semantics, consider a specification for a small expression language. The language is to support nested environments and exceptions. These are common language features, and from a standard library of monad transformers we select $ErrorT$ for supporting errors and $EnvT$ for environments. A monad for our expression language can be formed by composing these monad transformers as follows[1]:

$$\mathbf{type}\, M \quad = \quad ErrorT\ (EnvT\ Id)$$

Next, consider the equation for the addition operation:

$$\begin{array}{l} evaluate\ (Add\ e_1\ e_2) = \\ \quad evaluate\ e_1 >\!\!>\!= \lambda v_1 \rightarrow \\ \quad evaluate\ e_2 >\!\!>\!= \lambda v_2 \rightarrow \\ \quad return\ (v_1 + v_2) \end{array}$$

Here we simply evaluate the two subexpressions and return the sum of the results. Errors and environments are transparently passed around by the $>\!\!>\!=$ operator, and hence need not be considered at all by this equation.

The equation for a variable reference involves both environments and errors, and demonstrates the use of the operators provided by those monad transformers:

$$\begin{array}{l} evaluate\ (Var\ x) = \\ \quad rdEnv >\!\!>\!= \lambda\rho \rightarrow \\ \quad \mathbf{case}\ lookup\ x\ \rho\ \mathbf{of} \\ \qquad Just\ v \rightarrow return\ v \\ \qquad Nothing \rightarrow raise\ \text{“unbound identifier”} \end{array}$$

Here $rdEnv$ and $raise$ are operators provided by the environment and error monad transformers, respectively. The function $lookup$ is a helper function defined elsewhere. The environment is accessed by means of the operator $rdEnv$, “read the value of the environment”, which uses the environment that is passed implicitly within the monad; it does not need to appear explicitly as a parameter to $evaluate$.

Underneath this example, of course, are the actual monad transformers involved. As mentioned before, a monad transformer adds new operators to the monad, modifies the existing ones, and alters the definitions of $return$ and $>\!\!>\!=$. The definition of $EnvT$, the environment monad transformer, is as follows:

$$\begin{array}{rcl} \mathbf{type}\, EnvT\ m\ a &=& e \rightarrow m\ a \\ return_{(EnvT\ m)}\ v &=& \lambda\rho \rightarrow return_m\ v \\ c >\!\!>\!=_{(EnvT\ m)} f &=& \lambda\rho \rightarrow c\rho >\!\!>\!=_m \lambda v \rightarrow f v \rho \\ rdEnv_{(EnvT\ m)} &=& \lambda\rho \rightarrow return_m\ \rho \\ inEnv_{(EnvT\ m)}\ c &=& \lambda\rho' \rightarrow c\rho \\ lift_{(EnvT\ m)}\ c &=& \lambda\rho \rightarrow c \end{array}$$

[1] Id is the trivial monad

The first equation defines the new monad transformer $EnvT$ as taking a monad m defined over a result type a and turning it into a new monad, $EnvT\ m$, over a. An encoding in the lambda calculus is given for environments.

Next we define the operators $return_{(EnvT\ m)}$ and $>\!\!>\!=_{(EnvT\ m)}$, in terms of the operators of the lower monad m.

We define the operators $rdEnv$ and $inEnv$ next, thus permitting access to the environment now being passed around within the monad.

Finally, we define an operator $lift_{(EnvT\ m)}$ which is used to transform computations in the lower monad m to computations in the upper monad $EnvT\ m$. This lifting operator is used by the MMS system to lift the operators of m to the upper monad level, so they may still be used.

In reality, the situation is a little more complex than this. For details, consult Liang, Hudak, and Jones’ paper [LHJ95].

4 Modular monadic action semantics

We have seen that action semantics is an excellent notation for describing the semantics of real programming languages. However, we have also seen that certain constructs pose grave difficulties when the language specifier attempts to encode them in action semantics. These difficulties are due to the fixed nature of action semantics—certain notions of computation are built into action semantics, and any that are not must be tediously simulated. This situation is inadequate.

As we noted in Section 2.3, the fixed nature of action semantics is in large part due to the monolithic nature of its underlying semantic definition, written in a variant of Plotkin’s structural operational semantics.

The recent work on modular monadic semantics, described in Section 3, suggested to us a solution to the problem. Modular monadic semantics provides a mode of semantic definition that is truly modular. Yet it is also sufficiently low-level to be used for the specification of action notation. A tiered system with action notation specified by a modular monadic semantics would preserve the user-friendliness of action notation, but permit the notation to be modified relatively easily to incorporate even quite major modifications or additions to the notions of computation represented.

As an added bonus, modular monadic semantics descends ultimately from denotational semantics, and so inherits its rich theory—yet without the customary tangle of unmaintainable equations. Hence, while providing a clear operational interpretation of action notation, a modular monadic semantics for action notation would also provide a sound basis for the development of theories of action semantics.

The result of these observations is MMAS—a modular monadic action semantics. MMAS appears identical to Mosses' action semantics, but internally its semantics is specified by means of a modular monadic semantics in the style of Liang, Hudak, and Jones, rather than Mosses' structural operational semantics. As a consequence, MMAS is modular and extensible, and dialects of MMAS can be created that incorporate new or modified notions of computation.

4.1 Structure of MMAS

In Figure 1 we have considered the structure of action semantics as a formal specification technique, along with other semantic formalisms. We now consider the structure of systems that *implement* action semantics, in order to interpret or compile a language defined in it.

In Figure 2(a) we have Mosses' definitive formal model, as described in [Mos92]. The action-semantic interface of the system is described by a kernel action notation (here denoted 'KAN') and a layer of "sugar" reducing full action notation to this kernel action notation. The kernel is described in terms of a structural operational semantics (see Section 2.2), and this structural operational semantics is executed by an abstract machine.

It is not practical to implement Mosses' formal model directly. Instead such a system is simulated by some other technique, and this shown formally or informally to be equivalent to Mosses' scheme. The architecture of an action semantics interpreter of this nature is depicted in Figure 2(b). Notice that the interpreter in this system is generally more or less monolithic, and implemented in some low-level implementation language such as C.

An action compiler (or action semantics-based compiler generator) is similar, except that actual code generation is usually deferred to another compiler and hence the compiler is really a translator between action notation and the low-level implementation language. Figure 2(c) depicts this scenario. Again, the system is essentially monolithic: the program implements the semantics of action notation, but in an opaque manner that is not at all easy to modify.

In contrast to these techniques, consider the MMAS approach, as shown in Figure 2(d). Here the external action semantic interface of the system is provided by the *action notation layer*, which is coded in modular monadic semantics. The action notation layer depends on a *monad transformer layer*, which defines and combines a series of monad tranformers representing features required by the action notation layer. As there is some common code in the monad transformer layer, this is abstracted out into the *general monad transformer layer*; both these latter two layers

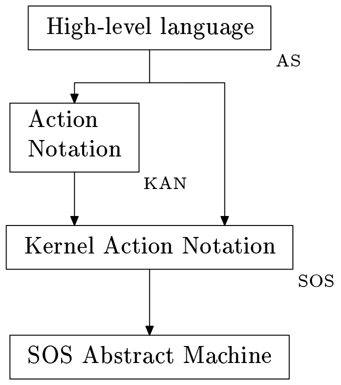

Figure 2(a): Mosses' model action semantics system

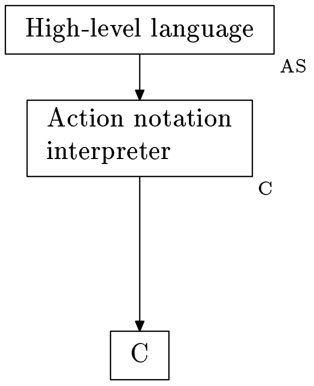

Figure 2(b): An action semantics interpreter

are coded in Haskell and the system depends on an underlying Haskell interpreter or compiler.

The larger number of distinct layers in this scheme ensures that flexibility and modularity is possible at each level. The modular structure of each layer is depicted in Figure 3. Features in the high-level language are based upon notions of computation provided by modules in the action notation layer, which are based in turn on those provided by the monad transformers of the monad transformer layer, and so on. As features change in the high-level language, the architecture allows existing modules to be removed or modified and additional modules to be added, with a high-degree of independence from other modules in the system.

4.2 The base system

Like Mosses' action semantics, modular monadic action semantics can be divided into facets. In general, each facet consists of an action notation module and a supporting monad transformer providing the notions of computation on which it relies. However, this need not be true of all facets: action notation modules and monad transformers

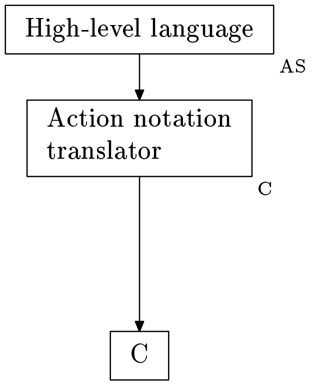

Figure 2(c): An action semantics compiler

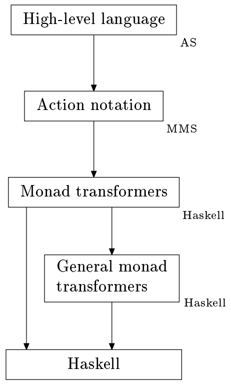

Figure 2(d): The MMAS action semantics system

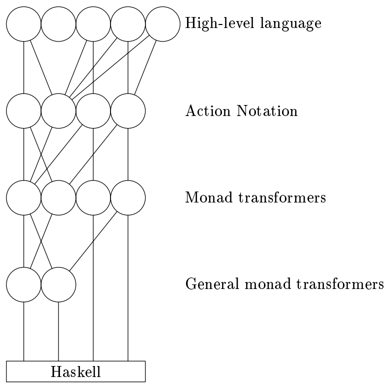

Figure 3: Detail of the MMAS action semantics system

are independent. Both an action notation module relying only upon existing monad transformers and a monad transformer with no related action notation module are possible, and indeed are present in base MMAS.

Base MMAS, the unmodified form of the MMAS system, implements almost all of Mosses' action semantics. It contains six action notation modules: basic and functional, declarative, imperative, reflective, directive, and communicative. These correspond directly to the seven facets of Mosses' action notation, except that the basic and functional facets have been merged into one.

To support this action notation layer, base MMAS contains eight monad transformers: one each for the basic and functional, declarative, imperative, directive, and communicative facets (there is none for the reflective facet), along with two others used to implement parallelism and nondeterminism (built into Mosses' structural operational semantics but not present in the lambda-calculus basis of MMAS).

Code common to a number of these monad transformers suggested the extraction of two general monad transformers, one for the representation of environments and the other for the representation of state.

This base system implements almost all of Mosses' action semantics: in most cases existing action semantic descriptions can be used with MMAS with very little modification. Such ASDs can be interpreted in an MMAS system based over a Haskell interpreter, or made into compilers using an optimising Haskell compiler. Proofs of their properties may take advantage of the true modularity provided by the modular monadic definition of action semantics (see [LH96] for a discussion of proofs in a modular monadic context).

4.3 Branching out

The key feature of modular monadic action semantics, absent in other implementations of action semantics, is *extensibility*. Modules can be modified, removed from or added to the MMAS system.

Modification Any module of the MMAS system can be modified internally without affecting any other portion of the system, as long as its external interface is not changed. In fact, new functionality may be *added*, as long as existing functionality is unaffected.

At the level of the action notation layer, new actions, combinators, yielders or sorts may be added or the precise behaviour of existing ones altered, simply by editing the modular monadic semantics code that specifies them for the facet concerned.

At the level of the monad transformer, new features may be added or the implementation of existing ones may be altered. New or altered features can then be used by the action notation module or modules above it.

As long as modifications preserve or naturally extend existing functionality, changes are completely modular. If this is not the case, changes are required only to the modules that depend on the module modified: in the case of changes to an action notation module, none (other than existing ASDs); in the case of changes to a monad transformer, only the action notation module(s) on which it depends.

Removal Naturally, modules which are not required may be removed from the MMAS system entirely. This is of course not strictly necessary—optimisations will remove reference to the unused module anyway, and proofs are modular and will not be affected by an unreferenced module—but may be desired for neatness or security.

Addition Of greater significance is the addition of new modules. It is clear that modifying existing facets or modules is not always sufficient; in certain cases one wishes to add an entirely new feature or notion of computation to the system. The procedure for doing so in MMAS (in contrast to that for existing action semantics systems) is straightforward.

First, a new monad transformer must be defined. This transformer is written in Haskell, possibly with the aid of the existing general monad transformers and support code, and encapsulates the behaviour of the new feature. A typical monad transformer can be defined in around 40 lines of code.

Next, a new module in the action notation layer is defined. This module is written using modular monadic semantics, referring to the new monad transformer and possibly others. It defines the new action notation to be used to access the newly-defined feature. Depending on how many actions, combinators and yielders are to be defined the code required varies, but a typical size would be around 30 lines of code.

Together, this module and its supporting monad transformer add the desired new feature to the MMAS system. The system can now be used in exactly the same manner as before—existing ASDs will continue to work identically, unaware of the new feature—but new action semantics descriptions may use the feature as desired.

4.4 Reasoning in MMAS

Consider again Figure 1, and the problem of reasoning about the correctness of a program or a transformation in the high-level language by means of the formal semantics.

In fig. 1 (a), a proof of a property of the high-level language is essentially a proof about states of the abstract machine upon which SOS is based. Similarly, in fig. 1 (b) a proof concerns elements of the various domains over which the relevant semantic equations are defined. In both cases, the representations we must deal with are very low-level, and *monolithic*: i.e., all information relating to the program is contained in a single object, and all of it can potentially affect the validity of the proof and must be taken into account by it. In both SOS and λ-calculus we may take steps to alleviate this to some extent, but it remains a fundamental problem.

In fig. 1 (c), Mosses' action semantics, *modular* proofs are possible at the action semantic level. This means that we may prove a property about, say, the functional behaviour of an expression without concerning ourselves with interactions from communication or from the store. Given an action theory describing the semantics of actions at the action semantics level, facet by facet, we can construct our proofs facet by facet also.

However, the assumption here is that such an action theory exists. Certainly this *should* be the case, but in general it is not. Certain properties about actions are known, and are listed in [Mos92] and elsewhere; but no general action theory yet exists (see Section 2.3). Because of this lack, proofs about high-level language properties must usually begin by constructing the action theory they need: and this construction of action theory must be performed at the SOS level, which as we saw above is monolithic rather than modular, and hence difficult. In practice, proofs about high-level languages using action semantics are almost as hard as those using a structural operational semantics directly.

The scenario in fig. 1 (d), however, is different. As in fig. 1 (c), we may construct modular proofs of high-level language properties based on the modularity of modular monadic semantics. But with modular monadic semantics, the theory we construct to support this may also be constructed modularly: our interaction with the lambda-calculus is structured in such a way that proofs involving one monad transformer cannot be affected by properties of other unrelated monad transformers. Hence we are able to realise the promise of relatively easy modular proofs of high-level language properties, even in practice when this requires the construction of new modular monadic semantic theory, since this construction is also modular.

As has already been explained, the intention of modular monadic action semantics (fig. 1(c)) is to provide action semantics with the true modularity of modular monadic semantics. To prove high-level language properties in this system, we use action theory. But, instead of constructing this theory directly and monolithically from the SOS, we construct it in terms of the relatively high-level modular theory provided by modular monadic semantics. *This* theory, in turn, is constructed, again modularly, from the lamba-calculus properties involved in each monad transformer. The result is a fully-modular, tiered system, in which one need only consider the features that are directly relevant to the property one wishes to prove—unrelated features may safely be ignored.

Work so far on modular monadic action semantics has concentrated rather on the pragmatics of the system than on the theory, but the theoretical basis is clear. Liang and Hudak [LH96] give a discussion of proofs in modular monadic semantics corresponding to the two arrows in that we used in fig. 1 (c) of Figure 1; these are essentially the lower two of the three arrows in fig. 1 (e), modular monadic action semantics.

4.5 Limitations

Modular monadic action semantics is not completely identical to Mosses' action semantics. MMAS implements only a small part of Mosses' communicative facet. Mosses' version of this facet supports general message-passing communication between multiple parallel agents. A system of this nature is of necessity quite complex, involving not merely communication but also the creation and management of the parallel processes themselves. As the whole area of parallelism in denotational semantics is currently rather turbulent [Abr96], it was felt safest to take a conservative approach and implement only a restricted form of process–user interaction, for a single process only. Parallelism within a process is supported.

The type system used by modular monadic action semantics is essentially that of the underlying Haskell system, overlaid with Liang, Hudak, and Jones' extensible union types [LHJ95]. Compared with the unified algebras used by Mosses, this system is quite restricted: types (*sorts* in Mosses' terminology) are not first-class, and the only type operations permitted are injection, projection and disjoint union. This means that those operators in action semantics dependent upon first-class sorts (such as the nondeterministic choice operator choose) have of necessity had their semantics altered. Luckily there are few of these: Mosses writes [Mos92, p. 36] "action notation does not depend much on the unorthodox features of our algebraic specification framework."

As well as being impacted by the lack of first-class sorts (and hence unbounded nondeterminism), the nondeterminism of modular monadic action semantics is restricted by its nature as an executable system. It is constrained to 'give an answer', and hence in the end just *one* of the many possible (nondeterministic) behaviours must be exhibited, unlike action semantics which merely returns the sort of all possible behaviours.

5 An example

The notion of computation that is most obviously missing from Mosses' action semantics is *first-class continuations* (noted in [Mos92, p. 211] and [Doh93, §4.1], amongst others). For various reasons, Mosses found continuations difficult and messy to add to action semantics, and so chose to omit them. Unfortunately, this makes a number of language constructs (including some forms of exception handling) exceedingly difficult to specify. Now, with MMAS providing the full power of denotational semantics, continuations can be implemented with relative ease.

Figure 4 shows the definition of the new monad transformer, $ContT$ (some details have been omitted). This monad transformer modifies $return$ and $>\!>=$ to use continuation-passing style, and defines an operator $callccK$ to perform the call-with-current-continuation operation. By adding this monad transformer to the monad transformers of base MMAS, all existing code will be transparently converted to use a continuation-passing semantics rather than a direct semantics. All existing monad operators are converted to behave appropriately, and the new $callccK$ operator is added.

Recall that at this point, the modularity of the system ensures that (as long as certain proof obligations are satisfied regarding the behaviour of $return_{(ContT\ m)}$, $>\!>=_{(ContT\ m)}$, and $lift_{(ContT\ m)}$) the behaviour of all existing action notation module code remains identical, and that all proofs of properties of the base MMAS system still remain valid for the new system. *No existing code* need be altered to support the use of this new notion of computation.

In order to use this new feature, however, new action notation must be provided. Figure 5 shows a portion of the code for the new action notation module: the definition of the combinator $cWithCC$ (for which the concrete syntax is with the current continuation in _ do _). In addition to this combinator (and omitted from the figure), two new actions are provided: jump to _ and jump to _ with _. The actions permit a continuation to be invoked, optionally passing it a value; the combinator binds the current continuation to a token in the environment and performs a block of code (the call-with-current-continuation operation, action-semantics style).

The precise semantics of the combinator can be seen by inspecting the modular monadic semantic code in Figure 5. The token yielder is evaluated, and then the $ContT$ operator $callccK$ is used to capture the current continuation. The token is bound to a code fragment that obtains the current transient, passes it to the continuation, obtains the return value and provides it as the transient result. Then the code enclosed by do _ is performed, the resulting transient obtained and returned. Finally, the result of $callccK$ is obtained and returned as a transient to the caller.

With the addition of the monad transformer of Figure 4 and the action notation module of Figure 5 are added, MMAS is extended to include continuations. ASDs can now be written that refer to continuations—see the example in the appendix.

But unlike the other systems depicted in Figure 1, the extension of MMAS has not changed the behaviour or interface of the system with respect to existing features. All existing ASDs will behave just as they did before, and code from them may be used within new ASDs without modifi-

```
type ContT ans m a     =  (a → m ans) → m ans

return_(ContT m) v     =  λk → k v
m >>=_(ContT m) f      =  λk → m (λa → f a k)

callccK_(ContT m) f    =  λk → f (λa → (λk' → k a)) k

lift_(ContT m) m       =  λk → m >>=_m k
```

Figure 4: Continuations: monad transformer

```
cWithCC ytok a =

 ytok >>= λ tok →
 callccK (λk →
   getE >>= λe →
   setE (overlay e
          (bindTo tok
             (abstractAct
                (getB >>= λ(Tr t) →
                 k t >>= λt' →
                 doB (Tr t'))))))    >>= λ() →
   a >>= λ() →
   getB >>= λb →
   return (case b of
              Tr t → t
              _    → tnil))    >>= λt →
 getB >>= λb →
 case b of
   Tr _ → doB (Tr t)
   _    → return ()
```

Figure 5: Continuations: action notation module portion

cation. Even more significantly, the modular proofs that applied to the old ASD and the old version of MMAS will apply equally well to the new extended version! No extra work is required—everything required is encapsulated within the new monad transformer, action notation module and proof obligations.

The full code for the continuation facet of MMAS and for all facets of base MMAS is given in Wansbrough's thesis [Wan97].

6 Related work

The work described here owes a great debt to the work of Mosses and others [Mos92, Mos96a] on action semantics.

Modular monadic semantics derives ultimately from the work of Moggi [Mog89a, Mog89b, Mog91b, Mog91a] on monads for programming language semantics. Steele's work [Ste94] on pseudomonads provided an early prototype of a system very similar to MMAS. Espinosa [Esp95] and Liang, Hudak, and Jones [LHJ95, LH96] provided the details of the implementation of Moggi's ideas in a working system; MMAS is directly based on the work of Liang, Hudak and Jones (the term 'modular monadic semantics' is due to Liang and Hudak [LH96]). It is interesting that both Espinosa and Liang, Hudak, and Jones credit Mosses with inspiring their research. This present paper exhibits a more concrete connection between the two groups.

A number of researchers have developed action interpreters or action compilers [Mou96, Ørb94, BMW92, Pal92], which appear similar to MMAS in that they implement action semantics; but in general these are based on Mosses' structural operational semantics and are constructed monolithically (see Figures 2(a), 2(b) and 2(c)).

Lassen[Las95] and Doh and Schmidt[DS94], like MMAS, replace Mosses' structural operational semantics with an alternative (a reduction semantics and a natural semantics, respectively) and use it to reason about action notation. Neither theory is intended to be executable, however, and neither is particularly modular.

7 Conclusions

In conclusion, then, action semantics is both an excellent system for the specification of domain-specific languages and a fascinating DSL in its own right. However, action semantics has significant limitations. It is incomplete and does not permit extension, and proving results within it is difficult. The solution to this problem is provided by modular monadic action semantics, a tiered system consisting of an action notation layer defined in terms of a modular monadic semantics, which is in turn modularly defined in terms of a functional programming language.

We have seen that modular monadic action semantics enhances action semantics, making it more useful and extending its range of applicability. Through its modularity and extensibility, new features can be added to the base action semantics, enabling the description of languages using these features to be done without resorting to tedious simulation. By replacing Mosses' structural operational semantics with a modular monadic one, we have in fact achieved what Mosses hints at in [Mos96a, §8], where he notes that "the current structural operational semantics of action notation is not so easy to modify; alternative forms ... might be preferable in that respect." We have demonstrated the utility of this by adding continuations to action semantics, in Section 5, something that has been until now quite impractical to achieve.

In addition, the use of a *modular* underlying semantics which is directly based on denotational semantics should make action semantic theory much easier to develop. Proofs will be modular, and can make use of the results and techniques that have been developed in the field of denotational semantics. Of course, as Mosses notes in [Mos96a, §1.3], there are certain technical difficulties with implementing *all* of action semantics in denotational semantics (and hence modular monadic semantics); however MMAS demonstrates that a substantial and useful portion of it *can* be so implemented. As further developments occur in denotational semantics, these may be brought into the MMAS framework and used to increase its scope.

The MMAS system demonstrates the utility of Liang, Hudak, and Jones' modular monadic semantics as a lower-level semantic framework. Our implementation consists of around 1200 lines of code, and so is a significantly-sized example of its use. We found that the system worked extremely well, although our experience did suggest some minor alterations to their approach.

Modular monadic action semantics is a flexible, modular, extensible version of Mosses' action semantics. It allows new features to be readily added to the semantics in a modular fashion, and promises to make the semantic theory more manageable. As such, we believe it offers an excellent extension to action semantics for specifying the semantics of domain-specific languages.

References

[Abr96] Samson Abramsky. Semantics of interaction. In Hélène Kirchner, editor, *Trees in Algebra and Programming—CAAP'96: Proceedings of the 21st International Colloquium, Linköping, Sweden, April 1996*, number 1059 in Lecture Notes in Computer Science, page 1. Springer-Verlag, 1996. Invited talk.

[BMW92] Deryck Brown, Hermano Moura, and David A. Watt. ACTRESS: an action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Compiler Construction: Proceedings of the 4th International Conference, CC'92, Paderborn, FRG, October 1992*, number 641 in Lecture Notes in Computer Science, pages 95–109. Springer-Verlag, 1992.

[Doh93] Kyung-Goo Doh. Action semantics: A tool for developing programming languages. In *Proceedings of InfoScience'93, International Conference on Information Science and Technology*, Seoul, Korea, 21–22 October 1993. Also available as Technical Report 93-1-005, The University of Aizu. Available `ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/Doh93IS/`.

[DS94] Kyung-Goo Doh and David A. Schmidt. The facets of action semantics: Some principles and applications (extended abstract). In Mosses [Mos94], pages 1–15.

[Esp93] David Espinosa. Modular denotational semantics. Unpublished manuscript, December 1993.

[Esp94] David Espinosa. Semantic Lego. Unpublished manuscript, January 1994. Available `http://www-swiss.ai.mit.edu/ftpdir/users/dae/`.

[Esp95] David A. Espinosa. *Semantic Lego*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1995. Available `http://www-swiss.ai.mit.edu/ftpdir/users/dae/`.

[FHK84] Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments: Report on a Workshop directed by F. L. Bauer and H. Remus*, volume 8 of *NATO ASI Series F: Computer and System Sciences*, pages 263–274. Springer-Verlag, 1984.

[HT94] B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In Mosses [Mos94], pages 34–42.

[Las95] Søren B. Lassen. Basic action theory. Technical Report RS-95-25, BRICS, Department of Computer Science, University of Aarhus, May 1995.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP '96: 6th European Symposium on Programming, Linköping, Sweden, April 1996*, number 1058 in Lecture Notes in Computer Science, pages 219–234. Springer-Verlag, 1996.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 22–25, 1995*, pages 333–343, New York, 1995. ACM Press.

[Mog89a] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1989. Available `http://theory.doc.ic.ac.uk:80/tfm/papers/MoggiE/abs-view.ps.gz`.

[Mog89b] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989.

[Mog91a] Eugenio Moggi. A modular approach to denotational semantics. In D. H. Pitt et al., editors, *Proceedings of Category Theory and Computer Science, Paris, France, September 3–6, 1991*, number 530 in Lecture Notes in Computer Science, pages 138–139. Springer-Verlag, 1991. Invited talk.

[Mog91b] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Mos92] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Mos94] Peter D. Mosses, editor. *Proceedings of the First International Workshop on Action Semantics, Edinburgh, 14 April 1994*, number NS-94-1 in BRICS Notes, 1994.

[Mos96a] Peter D. Mosses. Theory and practice of action semantics. In *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science, Cracow, Poland, September 1996*, number 1113 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Mos96b] Peter D. Mosses. A tutorial on action semantics. Tutorial notes for FME'96: Formal Methods Europe, Oxford, 18–22 March 1996, 1996. Draft. Available `ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/Mosses96DRAFT/`.

[Mou96] Hermano Moura. An implementation of action semantics (draft). Available `http://www.di.ufpe.br/~rat/`, 1996.

[NT94] J. P. Nielsen and J. U. Toft. Formal specification of ANDF, existing subset. Technical Report 202104/RPT/19, issue 2, DDC International A/S, Lundtoftevej 1C, DK-2800 Lyngby, Denmark, 1994.

[Ørb94] Peter Ørbæk. OASIS: An optimizing action-based compiler generator. In Peter A. Fritzson, editor, *Compiler Construction: Proceedings of the 5th International Conference, CC'94*, volume 786 of *Lecture Notes in Computer Science*, pages 1–15, Edinburgh, U.K., April 1994. Springer-Verlag.

[Pal92] Jens Palsberg. A provably correct compiler generator. In B. Krieg-Brückner, editor, *ESOP '92: Proceedings of the 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434, Rennes, France, February 1992. Springer-Verlag.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Department of Computer Science, University of Aarhus, 1981. Now available only from University of Edinburgh.

[Plo83] G. D. Plotkin. An operational semantics for CSP. In Dines Bjørner, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts—II, Garmisch-Partenkirchen, FRG, 1–4 June 1982*, pages 199–223, Amsterdam, 1983. North-Holland.

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, 1994. ACM Press.

[vDM96] Arie van Deursen and Peter D. Mosses. *ASD: The Action Semantic Description Tools User's Guide*, May 24 1996. Version 2.5. Available `http://www.brics.dk/Projects/AS/Tools/ASD/`.

[Wan97] Keith Wansbrough. A modular monadic action semantics. Master's thesis, Department of Computer Science, University of Auckland, February 1997. Available `http://www.cs.auckland.ac.nz/~kwan001/thesis/` .

A Devils and Angels

This is a problem from [FHK84]. Carlsson uses it as an example in his Advanced Functional Programming course at Chalmers. His solution code (in Haskell, using a primitive version of [LHJ95]'s monad transformer system) can be found at `http://www.cs.chalmers.se/~magnus/afp/problems/devils-n-angels/`; this code follows Friedman, Haynes and Kohlbecker's Scheme code in [FHK84].

The problem is to define three actions, milestone, devil and angel, with the following behaviour. The computation has the goal of finishing despite the existence of devils.

Whenever a devil is encountered, control is sent back to the last milestone. If another devil (or the same one again) is encountered, control is sent back to the milestone before that one, and so on. If no milestones remain, the devil does nothing.

Whenever an angel is encountered, control is sent forward to where the computation last met a devil. If another angel is encountered, control is sent further forward to the devil before that one, and so on. Again, if no devils have been encountered the angel does nothing.

The milestone appears as an action that simply passes a transient straight through, like regive. The value passed to a devil, however, is given to what follows the appropriate milestone; the value passed to an angel is given to what follows the appropriate devil.

Continuations provide an excellent means of implementing the above problem. We maintain two stacks of continuations: one of *past* continuations (pushed by milestones and popped by devils), and one of *future* continuations (pushed by devils and popped by angels). This is achieved by the following actions:

```
(1)  pop-cont $S$:Token =
        give the data stored in
             the cell bound to $S$ then
        | | check the count of it
        | |        is greater than 0 and then
        | | | | give the first of it
        | | | and
        | | | | store the rest of it in
        | | | |       the cell bound to $S$
        | or
        | | check the count of it is equal to 0 then
        | | | give the abstraction of regive .

(2)  push-cont ( $S$:Token, $Y$:Yielder ) =
        | | give $Y$
        | and
        | | give the data stored in
        | |       the cell bound to $S$
        then
        | store it in the cell bound to $S$ .
```

Note that the continuations are stored in a tuple stored in a named cell. We use cells named "past" and "future".

We can now define the required primitives, milestone, devil and angel, as follows:

```
(3)  milestone =
        with the current continuation in "k" do
        | | push-cont ( "past", the data bound to "k" )
        | and
        | | regive .

(4)  devil =
        with the current continuation in "k" do
        | | push-cont ( "future", the data bound to "k" )
        | and
        | | | pop-cont "past" and regive
        | | then
        | | | jump to the given continuation#1
        | | |        with the given datum#2 .

(5)  angel =
        | pop-cont "future" and regive
        then
        | jump to the given continuation#1
        |        with the given datum#2 .
```

milestone simply obtains the current continuation and pushes it onto the "past" stack, and then passes through the value passed to it.

devil obtains the current continuation, pushes it onto the "future" stack, and then passes the value passed it to the continuation popped off the top of the "past" stack (note that if the stack is empty, we are given the identity abstraction abstraction of regive, so we get the correct behaviour even in this case).

angel simply passes the value it is passed directly to the continuation popped from the top of the "future" stack. Again, if it is empty it uses the identity abstraction.

We conclude with code for a short example due to Carlsson:

```
(6)  supernatural =
        | allocate a cell then bind "past" to it moreover
        | | allocate a cell then bind "future" to it
        hence
        | | store () in the cell bound to "past" and
        | | | store () in the cell bound to "future"
        | then
        | | give 1 then
        | | | milestone then
        | | | | | check it is equal to 1 then
        | | | | | | give 2 then
        | | | | | | | devil then
        | | | | | | | | give the sum of ( it, 100 )
        | | | | or
        | | | | | check it is not equal to 1 and then
        | | | | | | give the sum of ( 3, it ) then
        | | | | | | | angel .
```

After allocating cells for the two stacks and initialising them, this code passes 1 to the first milestone; if it returns 1 then it passes 2 to a devil and gives a final result of whatever the devil returns plus 100. If the milestone doesn't return 1 then the code adds 3 to whatever the milestone did return and passes the result to an angel.

Execution proceeds as follows: 1 is passed to the milestone, and 1 is returned. 2 is passed to the devil, which jumps back to the milestone and returns 2 from it. 3 is added to 2 to get 5, which is passed to the angel. The angel jumps forward to the devil, which now returns 5; 100 is added to this to get 105, which is returned from the computation as the final result.