Action Semantics in Retrospect

David A. Watt

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. Email: `daw@dcs.gla.ac.uk`

Abstract. This paper is a themed account of the action semantics project, which Peter Mosses has led since the 1980s. It explains his motivations for developing action semantics, the inspirations behind its design, and the foundations of action semantics based on unified algebras. It goes on to outline some applications of action semantics to describe real programming languages, and some efforts to implement programming languages using action semantics directed compiler generation. It concludes by outlining more recent developments and reflecting on the success of the action semantics project.

1 Introduction

Action semantics arose out of Peter Mosses' dissatisfaction with existing semantic formalisms such as denotational semantics. He set out to develop a semantic formalism that would enable programming language descriptions to be comprehensible, modifiable, scalable, and reusable.

The action semantics project faced a number of challenges. It was necessary to design an action notation that would be powerful, natural, and yet manageable. Action semantics had to have solid foundations, in that the meaning of each action must be known precisely (and there must be no discrepancy between theory and intuition). Action semantics had to be tested in practice by writing descriptions of a variety of real programming languages.

All these challenges were addressed vigorously. Peter designed an action notation with a readable English-like syntax. He established its foundations by developing and using a new algebraic formalism called unified algebras. The operational flavour of action notation made it very natural for describing the dynamic semantics of programming languages (without forcing authors or readers to master the details of an abstract machine, as other operational formalisms do.) This same operational flavour also made action semantics directed compiler generation an attractive prospect.

A small but active action semantics community grew up during the 1980s and 1990s. To enable this community to keep in contact and exchange ideas, Peter organized a series of action semantics workshops [18, 23, 21].

The rest of this paper is structured as follows. Section 2 explains the dissatisfaction with denotational semantics (and other semantic formalisms) that motivated the development of action semantics. Section 3 recounts the early ideas and the eventual design of action semantics. Section 4 outlines the foundations of action semantics, in terms of the algebraic properties of action combinators, the new framework of unified algebras, and the operational semantics of action notation. Section 5 recounts some of the attempts to apply action semantics to the static and dynamic semantics of real programming languages. Section 6 briefly discusses some of the compiler-generation projects based on action semantics. Section 7 summarizes some more recent developments. Section 8 concludes by reflecting on the extent to which the action semantics project succeeded in its original aims. The appendices contain illustrative action semantic descriptions of three small programming languages, each built on its predecessor.

2 Motivations

As a research student at Oxford University in the 1970s, Peter gained a deep understanding of denotational semantics. He wrote a denotational description of Algol-60 [10]. He also developed his Semantic Implementation System (SIS) [11] which, given a denotational description of a programming language, would generate a compiler for that language. Later he wrote an overview of denotational semantics [16].

Experience shows that denotational descriptions have many pragmatic problems. Peter clearly perceived that the fundamental problem is the use of the lambda calculus as a notation for defining the semantic functions and auxiliary functions. Fundamental concepts such as binding, storing, sequencing, and choice must be *encoded* in the lambda calculus, rather than being expressed directly. Any change in the structure of a semantic function forces changes to all the semantic equations that define or use that semantic function.

To illustrate these points, suppose that we are developing a programming language incrementally. We will start with a simple applicative language APP; then add assignments to make a simple imperative language IMP; and finally add exceptions to make a language EXC.

If we are using denotational semantics to describe each successive language, the development might proceed as follows. When we describe the applicative language APP, the denotation of an expression is a function that maps an environment to a value. When we extend to the imperative language IMP, in which an expression may have side-effects, the denotation of an expression must map an environment and a store to a value and an updated store. When we extend to the language EXC, in which an expression might throw an exception, further wholesale changes are necessary, perhaps moving to the continuation style. In summary, the semantic functions for expressions change as follows:

(APP) $\mathit{eval}\ \_ : \mathit{Expression} \rightarrow (\mathit{Env} \rightarrow \mathit{Value})$
(IMP) $\mathit{eval}\ \_ : \mathit{Expression} \rightarrow (\mathit{Env} \rightarrow \mathit{Store} \rightarrow \mathit{Value} \times \mathit{Store})$
(EXC) $\mathit{eval}\ \_ : \mathit{Expression} \rightarrow (\mathit{Env} \rightarrow (\mathit{Value} \rightarrow \mathit{Cont}) \rightarrow \mathit{Cont})$

Even quite modest language extensions can force structural changes to the semantic functions, and hence changes to all the semantic equations defining or using these semantic functions. If the language were further extended to support concurrency, further wholesale changes would be needed, now employing the heavy machinery of power-domains.

A denotational description of a small programming language is feasible, but a denotational description of a realistic language on the scale of Pascal or Java constitutes a formidable challenge. In practice, nearly every published denotational description has been incomplete, typically omitting whatever the author deemed to be non-core features of the language.

Even experts find denotational descriptions hard to understand. To non-experts they appear unfamiliar, unnatural, and incomprehensible. For this reason, denotational descriptions have almost never been used in language reference manuals.

These pragmatic problems are not unique to denotational descriptions; in fact, they are shared by all the other well-known semantic formalisms. For example, in natural semantics [8], the designer who is developing a programming language incrementally must be prepared to change the structure of the judgements:

(APP) $e \vdash E \Rightarrow v$
(IMP) $e, s \vdash E \Rightarrow v, s'$
(EXC) $e, s \vdash E \Rightarrow v/x, s'$

(where E is an expression, e is an environment, v is a value, and v/x is either a value or an exception). Each change in a judgement forces changes to all semantic rules using that judgement.

So Peter set himself the challenge of devising a semantic formalism that would enable semantic descriptions to have good pragmatic properties. More precisely, semantic descriptions should be:

– comprehensible, i.e., able to be understood (at least at an intuitive level) by non-experts;
– modifiable, i.e., able to adapt to changes in the design of the described language, without disproportionate effort;
– scalable, i.e., able to grow proportionately to the size of the described language;
– reusable, i.e., able to be used (at least in part) in the description of a related language.

3 Inspirations

Peter's original idea (possibly inspired by Backus [1]) was to describe the semantics of a programming language's constructs using a set of abstract semantic entities, which he called *actions* [13, 14]. Each action represents a computation that receives and produces information. Simple actions represent elementary computations (such as binding an identifier to a value or storing a value in a cell). Compound actions can be composed from simple actions by means of *combinators*. Each combinator captures a standard form of control flow (sequencing, interleaving, or choice). Each combinator also captures a standard form of information flow (distribution, composition, etc.).

Thus Peter had already established the basis of what he later christened *action semantics*. A number of details still remained to be resolved. In particular, what set of simple actions and what set of combinators would be adequate to describe a variety of language constructs? Would a small number of simple actions and a small number of combinators be sufficient? What notation should be used to write down these simple actions and combinators? And would the resultant semantic descriptions truly be comprehensible, modifiable, scalable, and reusable?

Peter originally employed a very concise but cryptic notation for the actions. Later he replaced this with a more readable notation; for example, $\bigtriangledown$ became complete; '$A_1 \oplus A_2$' became 'A_1 or A_2', '$A_1 \otimes A_2$' became 'A_1 and A_2', and '$A_1 \odot A_2$' became 'A_1 then A_2'. Careful choice of names for primitives and combinators made it possible to write quite complex actions in an English-like notation that is comprehensible even to non-experts.

An important aspect of Peter's original idea was to factor the actions into *facets*. Each simple action has an effect in only one facet. A compound action may have an effect in more than one facet. Any action may be polymorphically extended to other (possibly unforeseen) facets, in which its behaviour is neutral. The significance of this is that it enables action terms (and hence semantic descriptions) to be modifiable and reusable.

An action can *complete*, *escape*, *fail*, or *diverge*.

The *basic facet* is concerned only with control flow. Exclusive choice, whereby only one of the sub-actions is performed, is exemplified by the combinator '_ or _'. Sequencing, whereby the second sub-action is performed only if and when the first sub-action has completed, is exemplified by the combinator '_ and then _'. Interleaving, whereby the sub-actions may be performed in any order, is exemplified by the combinator '_ and _'. Every combinator adopts one of these three patterns of control flow.

In the *functional facet* an action receives and produces *transient* data. The simple action 'give d' produces the datum yielded by d. The simple action 'check b', completes if b yields true but fails otherwise. These and some other simple actions contain terms called *yielders*, which allow these actions to use received information. The yielder 'the given s' yields the received transient datum (which must be of sort s). For example, 'give successor of the given integer' uses a received integer and produces a different integer.

In the *declarative facet* an action receives and produces *bindings*, which are associations between *tokens* and *bindable* data. The simple action 'bind k to d' produces a binding of the token k to the datum yielded by d. The yielder 'the s bound to k' yields the datum (of sort s) bound to the token k.

When an action is polymorphically extended to another facet, it produces no information in that other facet. For example, 'give 7' produces a single transient but no bindings; 'bind "n" to 3' produces a single binding but no transients; 'give the integer bound to "n"' uses a received binding and produces a transient. A compound action may produce both transients and bindings, for example, 'give 7 and bind "n" to 3'.

The idea that actions are polymorphically extensible to other facets enables a reasonably small set of combinators to describe the control flow and information flow of a variety of language constructs. The number of possible combinators is constrained by the fact that the information flow must be consistent with the control flow. In particular, an interleaving combinator cannot make transients or bindings flow from one sub-action to the other; a sequential combinator cannot make transients or bindings flow from the second sub-action to the first sub-action.

The combinator '_ and _', distributes received information and combines produced information. In 'A_1 and A_2', received transients and bindings are distributed to both A_1 and A_2, transients produced by A_1 and A_2 are combined by tupling, and bindings produced by A_1 and A_2 are combined by disjoint union. For example, if the compound action 'give successor of the given integer and bind "m" to the given integer' receives the transient value 5, both sub-actions will receive that 5, and the compound action will produce the transient 6 and a binding of "m" to 5.

The combinator '_ then _' behaves as functional composition in the functional facet. In 'A_1 then A_2', received transients are passed into A_1, transients produced by A_1 are passed into A_2, and transients produced by A_2 are produced by the compound action. In the declarative facet, '_ then _' distributes received bindings and combines produced bindings. For example, if the compound action 'give successor of the given integer then bind "m" to the given integer' receives the transient value 5, it will produce a binding of "m" to 6 (but will produce no transients).

Conversely, the combinator '_ hence _' behaves as functional composition in the declarative facet, but in the functional facet it distributes received transients and combines produced transients. This combinator captures the concept of scope: the bindings produced by A_1 are used in A_2, and nowhere else.

Remarkably, the handful of combinators already mentioned, plus a few others, turn out to be adequate to describe the vast majority of language constructs. There is very little need for more specialized combinators.

In the *imperative facet*, actions allocate, inspect, and update the store. The store is structured as a mapping from *cells* to *storable* data. Stored information is stable: updates cannot be forgotten, but can be superseded by later updates. The simple action 'allocate a cell' produces a previously-unallocated cell. The simple action 'store d in c' updates the cell yielded by c to contain the datum yielded by d. The yielder 'the s stored in c' inspects the cell yielded by c, which must contain a datum of sort s. Commands in imperative languages can be described very naturally by imperative actions.

In the *communicative facet*, a number of *agents* can be created, each charged with performing a particular action with its own local store. Agents send and receive messages to one another asynchronously. Communicated information is permanent: a message once sent can never be retrieved nor superseded. Communicative actions can be used to describe concurrent languages.

There are no combinators specifically associated with the imperative and communicative facets. Each combinator's behaviour in the basic facet controls the order in which imperative and communicative sub-actions are performed. In particular, '_ and _' and similar combinators allow compound imperative and concurrent sub-actions to be interleaved in a nondeterministic fashion.

The applicative language (APP)

Appendix A shows an action semantic description (ASD) of the applicative language APP.

Two semantic functions are introduced here. The semantic function 'evaluate _' maps each expression to an action that will produce a single value. The semantic function 'elaborate _' maps each declaration to an action that will produce bindings.

We see here the notation used for sorts of actions. The sort 'action' includes all actions. The subsort 'action [giving a value]' includes only those actions that produce a single value – this is the sort of 'evaluate E'. The subsort 'action [binding]' includes only those actions that produce bindings – this is the sort of 'elaborate D'.

The compound action 'furthermore A' overlays the received bindings by the bindings produced by A.

Functions in APP are modeled by *abstractions*, each of which encapsulates an action. The operation 'abstraction of A' creates an abstraction encapsulating the action A. The operation 'closure of a' injects the received bindings into the abstraction yielded by a (such that these bindings will be received by the encapsulated action when it is eventually performed). Similarly, the operation 'application of a to d' injects the datum yielded by d into the abstraction a. Finally, the simple action 'enact a' performs the action encapsulated by the abstraction a. The notation for subsorts of abstractions parallels that for subsorts of actions, so functions are modeled by abstractions of subsort 'abstraction [using the given value | giving a value]'; this means that each abstraction encapsulates an action that both receives a value and produces a value.

The imperative language (IMP)

We can easily extend our applicative language to an imperative language IMP, whose ASD is shown in Appendix B.

Each expression is now mapped to an action of sort 'action [giving a value | storing]', which includes those actions that both produce a value and (potentially) update the store. This reflects the fact that expressions now have side-effects.

Each declaration is now mapped to an action of sort 'action [binding | storing]', which includes only those actions that produce bindings and update the store. This reflects the fact that declarations now have side-effects (in particular, variable declarations create and initialize cells). Semantic equations have been added for the new constructs: assignment expressions and variable declarations.

Despite the changes to the denotations, very little change is needed to the existing semantic equations. The semantic equation for 'evaluate I' must be modified to take into account of the fact that in IMP the identifier I could be bound to a cell. None of the other semantic equations need be changed. For example, in 'evaluate $[\![$ E_1 "+" E_2 $]\!]$', the sub-expressions E_1 and E_2 are meant to be evaluated collaterally, and for this the combinator '_ and _' is still appropriate. (On the other hand, if we decided that the sub-expressions should be evaluated sequentially, we would simply replace the '_ and _' combinator by the '_ and then _' combinator.)

The imperative language with exceptions (EXC)

We can further extend the language to allow expressions and declarations to throw exceptions, leading to the language EXC whose ASD is shown in Appendix C.

Each expression is now mapped to an action of sort 'action [giving a value | storing | escaping with an exception]', which includes only actions that either complete giving a value or escape giving an exception, updating the store in either case. The denotations of declarations have been changed likewise. Despite these changes to the denotations, *none* of the existing semantic equations needs to be changed. Semantic equations have been added for the new constructs: expressions that throw and catch exceptions.

The simple action 'escape with d' escapes producing the datum yielded by d as a transient. When an action escapes, enclosing actions escape too, except where the '_ trap _' combinator is used. The action 'A_1 trap A_2' will perform A_2 if and only if A_1 escapes, in which case any transients produced by A_1 are received by A_2.

The combinators used in the ASD of IMP are still appropriate in the ASD of EXC. For example, the semantic equation for 'evaluate $[\![$ E_1 "+" E_2 $]\!]$' still works because the '_ and _' combinator causes the compound action to escape if either sub-action escapes.

4 Foundations

The pragmatic benefits of action semantics would amount to little if the action notation were not well-founded. Peter addressed this problem with his usual energy and his deep understanding of the foundations of computation. He tackled the problem at three levels.

Algebraic properties

The action combinators have a number of simple algebraic properties, unsurprisingly. For example, the combinator '_ or _', is total, associative, and commutative, and has a simple action (fail) as its unit; similarly, the combinators '_ and _', '_ and then _', '_ then _', and '_ hence _', are all total and associative, and each has a simple action as its unit.

These algebraic properties can be used to reason about compound actions. They are insufficient to define the meanings of these actions precisely. However, they are sufficient to reduce the whole of action notation to a moderately-sized kernel. For full details see Appendix B of Peter's book [17].

Unified algebras

Peter invented a wholly new formalism of *unified algebras*. As its name suggests, this formalism is characterized by a unified treatment of ordinary values (*individuals*) and sorts (*choices*). Each individual is just a singleton sort. Each operation maps a choice to a choice. Unified algebras are in fact an independent contribution to formal methods, and applicable well beyond action semantics.

The clause:

- truth-value = true **|** false (*individual*) .

defines truth-value to be the choice between the individuals true and false. The further clauses:

- not _ : truth-value → truth-value (*total*) .
- not false = true ; not true = false .

introduce the usual 'not _' operation.

A more interesting example is:

- 0 : natural .
- successor of _ : natural → natural (*total*) .
- natural = 0 **|** successor of natural .

which defines natural to be the choice among the individuals 0, successor of 0, successor of successor of 0, etc. The operation 'successor of _' maps any choice of natural numbers to the choice of their successors; in particular, 'successor of natural' maps the natural numbers to the positive integers.

In this way Peter defined the *data notation* that underlies action notation. Data notation includes partial and total orders, tuples, truth-values, numbers, characters, lists, strings, syntax, sets, and maps. For full details see Appendix E of [17].

The author of an ASD of a particular programming language also uses unified algebras to define sorts (such as bindable and storable) that are used but not defined by action notation; and to define sorts specific to the described language.

Semantics of action notation

Peter also used unified algebras to define *action notation* itself. We can now see that action is the choice among all actions, and that action[...] is a restriction of that choice. We have already seen examples such as 'action [giving a value]', 'action [binding]', and 'action [binding | storing]'.

The meanings of actions (and yielders) in the kernel of action notation are defined using structural operational semantics (SOS) [26]. This handles all facets, including the communicative facet. It is a 'small-step' semantics, and correctly defines interleaving and nondeterminism. A step in a sequencing action such as 'A_1 and then A_2' is a step in A_1 (unless A_1 has terminated). A step in an interleaving action such as 'A_1 and A_2' can be a step in either A_1 or A_2 (unless one of these sub-actions has terminated). A step in an exclusive choice action such as 'A_1 or A_2' can be a step in either A_1 or A_2 (unless one of these sub-actions has committed – i.e., has performed an irreversible step such as updating the store or sending a message – in which case the other sub-action is abandoned).

The SOS of action notation is expressed in the notation of unified algebras. Although unified algebra operations are functions, their results can be *choices*. This allows nondeterminism to be expressed. Consider, for example, the operation that performs a single step in the action 'A_1 and A_2'; its result is the choice of configurations that can arise from performing *either* a single step in A_1 *or* a single step in A_2.

For full details see Appendix C of [17].

5 Applications

To test the pragmatic properties of action semantics, it was necessary to write ASDs of real programming languages. The first such test was an ASD of Pascal developed by Peter and me [22]. This was followed by ASDs of various imperative, object-oriented, functional, and concurrent programming languages, including CCS and CSP [4], Java [3], and Standard ML [28]. In his book [17], Peter shows the incremental development of an ASD of a substantial subset of Ada, including tasks.

These tests convincingly demonstrated that action semantics does indeed have the desired pragmatic properties. It is feasible to build complete ASDs of real programming languages. It is possible to reuse parts of these ASDs to describe related languages, even where the successor language is significantly richer than its predecessor. (For example, I reused large parts of the Pascal ASD in an ASD for Modula-3, which extends Pascal with objects, exceptions, and concurrency.) The ASDs are comprehensible and (in the usual sense of the word) natural – certainly far more so than corresponding denotational or natural semantic descriptions would be.

Although action notation was originally designed to describe dynamic semantics, Peter and I found that it can also be used to describe static semantics. This idea was tested on Pascal [22] and Standard ML [28]. Again, these tests demonstrated that action semantics is usable. In the Pascal static semantics, the enforcement of nominal type equivalence required the creation of a new type at each type-denoter, and this was encoded (rather unnaturally, it must be admitted) using the imperative facet. The Pascal static semantics thus reads like a type-checking algorithm. Likewise, in the first version of my ML static semantics, an auxiliary action ‘unify _ with _’ encoded the unification algorithm. A later version of my ML static semantics used unified algebra notation rather than action notation. The semantic function for each expression yields a *choice* of types – the expression’s principal type and all its instances. This gave the ML static semantics a more natural relational flavour.

6 Implementations

As we have seen, Peter started his academic career by building a compiler generator, SIS, based on denotational semantics [11]. Later he addressed the problem of compiler correctness [12].

Action notation has an operational flavour (unlike the lambda calculus). It directly supports the basic computational concepts of ordinary programming languages (rather than encoding them in terms of mathematical abstractions). So it is natural to think of generating compilers from action semantics.

An ASD of programming language L can be seen as specifying a translation from the abstract syntax of L to action notation. If a parser for L is composed with the L-to-action-notation translator and an interpreter for action notation, the product is an interpretive compiler for L. If a parser for L is composed with the L-to-action-notation translator and a code generator for action notation, the product is a full compiler for L.

Several action-notation interpreters have been written. My first interpreter, written in ML, covered the functional, declarative, and imperative facets of the action notation kernel, but did not support nondeterminism, interleaving, or the communicative facet. Hermano Moura’s interpreter, also written in ML, was more general. Later I wrote an action-notation interpreter in ASF+SDF, which did support nondeterminism.

In action notation the imperative facet is well-behaved, since store updates are stable. (This differs from denotational semantics, where stores are variables like any other, so stores can be cloned and updates can be reversed.) Overall, action notation behaves sufficiently like conventional imperative code to make it feasible to translate action notation to reasonably efficient object code.

Actress [2] was a compiler generator based on a large subset of action notation (excluding mainly the communicative facet). Given the ASD of any programming language L, Actress could generate a compiler composed of:

– a parser, which was generated from L’s syntax, and which translated the source code to an abstract syntax tree (AST);
– an AST-to-action-notation translator, which was generated from L’s semantics;
– an action-notation sort-checker, which inferred the sorts of data in the functional and declarative facets;
– an action-notation transformer, which performed various transformations of the action notation, including algebraic simplifications, bindings elimination, and stack storage allocation;

– an action-notation code generator, which translated action notation to object code expressed in C.

Actress could generate a better-quality compiler if L was statically-scoped and statically-typed, but it did not insist on these properties.

Cantor [25] was a compiler generator based on a broadly similar subset of action notation. Given the ASD of a statically-scoped and statically-typed language L, Cantor could generate a compiler composed of:

– a parser, which was generated from L's syntax, and which translated the source code to an AST;
– an AST-to-action-notation translator, which was generated from L's semantics;
– an action-notation code generator, which translated action notation to idealized SPARC assembly code;
– an assembler, which translated the idealized SPARC assembly code to real machine code.

Significantly, Cantor came with a correctness proof, which verified *inter alia* that the generated compiler would translate statically-typed source code to object code that will not misbehave despite being untyped.

Oasis [24] was a compiler generator that accepted an ASD expressed in Scheme syntax. Given the ASD of a statically-scoped and statically-typed language L, Oasis could generate a compiler composed of:

– a parser, separately generated from L's syntax (using YACC, say);
– an AST-to-action-notation translator, which translated the AST into Scheme syntax, combined it with the ASD already expressed in Scheme, and then interpreted the resulting Scheme program to generate action notation;
– an action-notation code generator, which translated action notation to SPARC assembly code;
– an assembler, which translated the SPARC assembly code to real machine code.

Oasis's code generator employed a variety of forward and backward analyses, which *inter alia* distinguished between bindings of known and unknown values, achieved constant propagation, and discovered opportunities for stack storage allocation. Oasis-generated compilers could generate remarkably efficient object code.

Because action notation directly reflects the fundamental concepts of programming languages, it seems to be feasible for a compiler generator automatically to discover key properties of the described language L, and to exploit these properties to improve the generated compiler and/or its object code. Here are some examples of such properties:

– Does L require stack allocation and/or heap allocation?
– Is L statically-scoped or dynamically-scoped?
– Is L statically-typed or dynamically-typed?

It is quite easy for a compiler generator to determine whether L is statically-scoped or dynamically-scoped by analyzing the semantics of L. If bindings are always injected into an abstraction (by means of the 'closure of _' operation) when the abstraction is created, L is statically-scoped. If bindings are always injected into an abstraction when the abstraction is enacted, L is dynamically-scoped.

It is rather more difficult for a compiler generator to determine whether the described language L is statically-typed or dynamically-typed. If both a static semantics and a dynamic semantics of L are provided, these would have to be analyzed together. A simpler approach is to focus on L's dynamic semantics, which invariably contains implicit sort information. (For example, consider the semantic equations for 'evaluate $[\![$ E_1 "=" E_2 $]\!]$' and 'evaluate $[\![$ "if" E_1:Expression "then" E_2:Expression "else" E_3:Expression $]\!]$' in Appendix A.) Analysis of the ASD opens up the prospect of automatically extracting type rules and determining whether L is statically-typed. The simplest approach of all is sort inference on a program-by-program basis, which is essential if the generated compiler is to generate efficient object code. The richness and generality of unified algebras make all these approaches challenging, but they have been extensively studied, most notably by David Schmidt's group [6, 7, 27].

7 Later developments

Modular structural operational semantics

SOS suffers from the same pragmatic problems as other semantic formalisms: any change in the structure of the configurations forces a major rewrite. The SOS of action notation was not immune to these problems.

Peter, with his characteristic energy, developed what he called *modular structural operational semantics* (MSOS) [19]. MSOS abstracts away from the structure of the configurations in somewhat the same manner as action semantics abstracts away from the structure of denotations. Peter then used MSOS to rewrite the semantics of action notation [20].

Action notation revisited

When the design of action notation was complete, it turned out to be larger than anticipated. In particular, there were fifteen action combinators in the kernel action notation (although six of these were rarely used). Moreover, there was an overlap between yielders and functional actions: both receive information and produce data. There was duplication between actions and abstractions: the action and abstraction subsort notations were similar, and for each action combinator there was a corresponding (but rarely-used) abstraction combinator. The communicative facet was complicated, and yet made it difficult to express the semantics of threads.

Working with Søren Lassen, Peter took a fresh look at action notation. They removed yielders from the kernel, but kept them in the full notation for the sake of fluency. They allowed actions to be treated as data, thereby eliminating the separate notion of an abstraction (and its associated notation). They also removed a number of features that were rarely-used or of doubtful utility. They completely redesigned the communicative facet, most significantly allowing agents to share a global store.

The resulting proposed version of action notation, *AN-2* [9], turned out to be about half the size of its predecessor, and its kernel was slightly smaller too.

Reusing programming language descriptions

One of the main aims of action semantics was reuse of programming language descriptions, particularly when describing a new language that inherits features from an older language. This has been demonstrated informally, as we saw in Section 5.

Working with Kyung-Goo Doh, Peter set out to demonstrate this more formally [5]. Their idea was to build a library of common programming language constructs, in which each module essentially contains a single semantic equation. For example, we could have a module for while-commands, a module for procedure calls with call-by-value semantics, a module for procedure calls with call-by-name semantics, and so on. Building an ASD of a new programming language then consists largely of selecting the appropriate modules from the library. Any conflicts that might arise (for example, if both call-by-value and call-by-name semantics are selected and they are not syntactically distinguished) can be detected readily.

8 Conclusion

Has the action semantics project been successful?

In theoretical terms, the answer is unequivocally yes. Action semantics has proved to be powerful enough to describe a large variety of programming language constructs. Action-semantic descriptions never rely on semi-formal notational conventions (unlike most denotational and natural semantic descriptions). Action notation has a secure foundation, and the drive to provide a secure foundation has produced remarkable spin-offs in unified algebras and MSOS. Another spin-off has been that the facet structure of action notation provides an intellectual framework for understanding the fundamental concepts of programming languages [29].

In practical terms, the answer is more equivocal. Action semantics has made it possible, for the first time, to write semantic descriptions of real programming languages that are comprehensible (even to non-experts), modifiable, scalable, and reusable. Action semantics directed compiler generation has been demonstrated to be feasible. However, engineering a high-quality compiler generator is a long-term project, and the necessary sustained effort has not yet been achieved. The action semantics community grew rapidly and spread over five continents, but it never exceeded 20 researchers at any one time, not enough to secure a long-term critical mass of activity.

I would like to take this opportunity to acknowledge the many contributions of the action semantics community, and above all to acknowledge the energetic leadership of Peter Mosses. This paper has set out to be an impressionistic overview rather than a comprehensive account of the action semantics project. Almost certainly it does not do justice to everyone's contribution, for which I can only apologize.

References

1. J. Backus (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Communications of the ACM 21*, pp. 613–641.
2. D.F. Brown, H.P. de Moura, and D.A. Watt (1990) Actress: an action semantics directed compiler generator, in *Compiler Construction: 4th International Conference* (ed. U. Kastens and P. Pfahler), pp. 95–109, Springer-Verlag.
3. D.F. Brown and D.A. Watt (1999) JAS: a Java action semantics, in [23], pp. 43–56.
4. S. Christensen and M.H. Olsen (1988) Action semantics of CCS and CSP, Report DAIMI IR-44, Computer Science Department, Aarhus University.
5. K.-G. Doh and P.D. Mosses (2003) Composing programming languages by combining action-semantics modules, BRICS Report Series RS-03-53, Computer Science Department, Aarhus University.
6. K.-G. Doh and D.A. Schmidt (1992) Extraction of strong typing laws from action semantics definitions, in *European Symposium on Programming 1992*, pp. 151–166, Springer-Verlag.
7. S. Even and D.A. Schmidt (1990) Type inference for action semantics, in *European Symposium on Programming 1990*, pp. 71–95, Springer-Verlag.
8. G. Kahn (1987) Natural semantics, in *Proceedings of STACS 1987*, pp. 22–39, Springer-Verlag.
9. S. Lassen, P.D. Mosses, and D.A. Watt (2000) an introduction to AN-2: the proposed new version of action notation, in [21], pp. 19–36.
10. P.D. Mosses (1974) The mathematical semantics of Algol60, Technical Monograph PRG-12, Programming Research Group, Oxford University.
11. P.D. Mosses (1975) Mathematical semantics and compiler generation, DPhil thesis, Oxford University.
12. P.D. Mosses (1980) A constructive approach to compiler correctness, in *International Colloquium on Automata, Languages, and Programming*, Springer-Verlag.
13. P.D. Mosses (1983) Abstract semantic algebras!, in *Formal Description of Programming Concepts II* (ed. D. Bjørner), North-Holland.
14. P.D. Mosses (1984) A basic abstract semantic algebra, in *International Symposium on Semantics of Data Types*, Springer-Verlag.
15. P.D. Mosses (1989) Unified algebras and action semantics, in *Proceedings of STACS 1989*, pp. 17–35, Springer-Verlag.
16. P.D. Mosses (1990) Denotational semantics, in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen *et al.*), pp. 575–631, Elsevier.
17. P.D. Mosses (1992) *Action Semantics*, Cambridge Tracts in Theoretical Computer Science.

18. P.D. Mosses (ed.) (1994) *First International Workshop on Action Semantics*, BRICS Notes Series NS-94-1, Computer Science Department, Aarhus University.
19. P.D. Mosses (1999) Foundations of modular structural operational semantics, in *Mathematical Foundations of Computer Science 1999*, Springer-Verlag.
20. P.D. Mosses (1999) A modular SOS for action notation, in [23], pp. 131–142.
21. P.D. Mosses and H.P. de Moura (eds.) (2000) *Third International Workshop on Action Semantics*, BRICS Notes Series NS-00-6, Computer Science Department, Aarhus University.
22. P.D. Mosses and D.A. Watt (1993) Pascal: action semantics, version 0.6 (unpublished).
23. P.D. Mosses and D.A. Watt (eds.) (1999) *Second International Workshop on Action Semantics*, BRICS Notes Series NS-99-3, Computer Science Department, Aarhus University.
24. P. Ørbæk (1994) Oasis: an optimizing action-based compiler generator, in *Compiler Construction: 5th International Conference* (ed. P.A. Fritzson), pp. 1–15, Springer-Verlag.
25. J. Palsberg (1992) A provably correct compiler generator, in *European Symposium on Programming 1992*, pp. 418–434, Springer-Verlag.
26. G.D. Plotkin (1981) A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Department, Aarhus University.
27. D.A. Schmidt and K.-G. Doh (1994) The facets of action semantics - some principles and applications, in [18], pp. 1–15.
28. D.A. Watt (1999) The static and dynamic semantics of Standard ML, in [23], pp. 155–172.
29. D.A. Watt (2004) *Programming Language Design Concepts*, Wiley.

A ASD of the applicative language (APP)

Semantic entities

- bindable = value **|** function .
- value = truth-value **|** integer .
- function = abstraction [using the given value **|** giving a value] .

Semantics of expressions

- evaluate _ : Expression → action [giving a value] .

```
(1) evaluate ⟦ $L$:Literal ⟧ =
        give the literal value of $L$ .
(2) evaluate ⟦ $I$:Identifier ⟧ =
        give the value bound to $I$ .
(3) evaluate ⟦ $E_1$ “+” $E_2$ ⟧ =
        | evaluate $E_1$ and evaluate $E_2$
        then give sum of (the given integer#1, the given integer#2) .
(4) evaluate ⟦ $E_1$ “=” $E_2$ ⟧ =
        | evaluate $E_1$ and evaluate $E_2$
        then give (the given value#1 is the given value#2) .
(5) evaluate ⟦ $I$ “(” $E$ “)” ⟧ =
        evaluate $E$ then
        enact application of (the function bound to $I$) to the given value .
(6) evaluate ⟦ “let” $D$:Declaration “in” $E$:Expression ⟧ =
        furthermore elaborate $D$
        hence evaluate $E$ .
(7) evaluate ⟦ “if” $E_1$:Expression “then” $E_2$:Expression “else” $E_3$:Expression ⟧ =
        evaluate $E_1$ then
        | | check the given value is true then evaluate $E_2$
        | or
        | | check the given value is false then evaluate $E_3$ .
```

Semantics of declarations

- elaborate _ : Declaration → action [binding] .

```
(1) elaborate ⟦ “val” $I$:Identifier “=” $E$:Expression ⟧ =
        evaluate $E$ then bind $I$ to the given value .
(2) elaborate ⟦ “fun” $I_1$:Identifier “(” $I_2$:Identifier “)” “=” $E$:Expression ⟧ =
        bind $I_1$ to closure of abstraction of
            | furthermore bind $I_2$ to the given value
            | hence evaluate $E$ .
```

B ASD of the imperative language (IMP)

Semantic entities

- bindable = value | function | cell .
- storable = value .
- value = . . .
- function = abstraction [using the given value | giving a value | storing] .

Semantics of expressions

```
•   evaluate _ : Expression → action [giving a value | storing] .
(1) evaluate ⟦ $L$:Literal ⟧ = . . .
(2) evaluate ⟦ $I$:Identifier ⟧ =
        give the value bound to $I$ or
        give the value stored in the cell bound to $I$ .
(3) evaluate ⟦ $E_1$ “+” $E_2$ ⟧ = . . .
(4) evaluate ⟦ $E_1$ “=” $E_2$ ⟧ = . . .
(5) evaluate ⟦ $I$ “(” $E$ “)” ⟧ = . . .
(6) evaluate ⟦ “let” $D$:Declaration “in” $E$:Expression ⟧ = . . .
(7) evaluate ⟦ “if” $E_1$:Expression “then” $E_2$:Expression “else” $E_3$:Expression ⟧ =
        . . .
(8) evaluate ⟦ $I$:Identifier “:=” $E$:Expression ⟧ =
        evaluate $E$ then
        | store the given value in the cell bound to $I$ and
        | give the given value .
```

Semantics of declarations

```
•   elaborate _ : Declaration → action [binding | storing] .
(1) elaborate ⟦ “val” $I$:Identifier “=” $E$:Expression ⟧ = . . .
(2) elaborate ⟦ “fun” $I_1$:Identifier “(” $I_2$:Identifier “)” “=” $E$:Expression ⟧ = . . .
(3) elaborate ⟦ “var” $I$:Identifier “:=” $E$:Expression ⟧ =
        | evaluate $E$ and allocate a cell
        then
        | store the given value#1 in the given cell#2 and
        | bind $I$ to the given cell#2 .
```

C ASD of the imperative language with exceptions (EXC)

Semantic entities

- bindable = value **|** function **|** cell **|** exception .
- storable = . . .
- value = . . .
- function = abstraction [using the given value **|** giving a value **|** storing **|** escaping with an exception] .
- exception $\leq$ distinct-datum .

Semantics of expressions

- evaluate _ : Expression $\rightarrow$ action [giving a value **|** storing **|** escaping with an exception] .

```
(1)  evaluate ⟦ L:Literal ⟧ = . . .
(2)  evaluate ⟦ I:Identifier ⟧ = . . .
(3)  evaluate ⟦ E1 "+" E2 ⟧ = . . .
(4)  evaluate ⟦ E1 "=" E2 ⟧ = . . .
(5)  evaluate ⟦ I "(" E ")" ⟧ = . . .
(6)  evaluate ⟦ "let" D:Declaration "in" E:Expression ⟧ = . . .
(7)  evaluate ⟦ "if" E1:Expression "then" E2:Expression "else" E3:Expression ⟧ =
         . . .
(8)  evaluate ⟦ I:Identifier ":=" E:Expression ⟧ = . . .
(9)  evaluate ⟦ "throw" I:Identifier ⟧ =
         escape with the exception bound to I .
(10) evaluate ⟦ E1 "catch" I:Identifier "then" E2 ⟧ =
         evaluate E1
         trap
         | | check the given exception is the exception bound to I
         | | then evaluate E2
         | or
         | | check not the given exception is the exception bound to I
         | | and then escape with the given exception .
```

Semantics of declarations

- elaborate _ : Declaration $\rightarrow$ action [binding **|** storing **|** escaping with an exception] .

```
(1)  elaborate ⟦ "val" I:Identifier "=" E:Expression ⟧ = . . .
(2)  elaborate ⟦ "fun" I1:Identifier "(" I2:Identifier ")" "=" E:Expression ⟧ = . . .
(3)  elaborate ⟦ "var" I:Identifier ":=" E:Expression ⟧ = . . .
```