BRICS

Basic Research in Computer Science

BRICS RS-96-53 P. D. Mosses: Theory and Practice of Action Semantics

Theory and Practice of Action Semantics

Peter D. Mosses

BRICS Report Series RS-96-53

ISSN 0909-0878 December 1996

Theory and Practice of Action Semantics

Peter D. Mosses[⋆]

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

Abstract. Action Semantics is a framework for the formal description of programming languages. Its main advantage over other frameworks is pragmatic: action-semantic descriptions (ASDs) scale up smoothly to realistic programming languages. This is due to the inherent extensibility and modifiability of ASDs, ensuring that extensions and changes to the described language require only proportionate changes in its description. (In denotational or operational semantics, adding an unforeseen construct to a language may require a reformulation of the entire description.)

After sketching the background for the development of action semantics, we summarize the main ideas of the framework, and provide a simple illustrative example of an ASD. We identify which features of ASDs are crucial for good pragmatics. Then we explain the foundations of action semantics, and survey recent advances in its theory and practical applications. Finally, we assess the prospects for further development and use of action semantics.

The action semantics framework was initially developed at the University of Aarhus by the present author, in collaboration with David Watt (University of Glasgow). Groups and individuals scattered around five continents have since contributed to its theory and practice.

1 Background

Readers of this paper are presumably familiar with the main ideas of denotational, operational, and axiomatic semantics. Action semantics was developed in response to problems with application of these frameworks—especially denotational semantics—to practical programming languages.

1.1 Some Problems with Denotational Semantics

First, a bit about my own background: I was fortunate to be studying at the University of Oxford when Scott and Strachey started their collaboration on the development of denotational semantics in 1969. I became an enthusiastic

⋆ E-mail address: `pdmosses@brics.dk`, WWW URL: `http://www.brics.dk/~pdm`

⋆⋆ Centre for Basic Research in Computer Science, Danish National Research Foundation.

follower of the approach, and the first paper I ever wrote provided a fairly complete denotational description of Algol60 [28]. My thesis work was on the use of denotational-semantic descriptions in compiler generation [29, 30], and I carried on to develop a prototype semantics implementation system called SIS [32].

SIS, which is now obsolete, took denotational descriptions as input. It transformed a denotational description into a λ-expression which, when applied to the abstract syntax of a program, reduced to a λ-expression that represented the semantics of the program in the form of an input-output function. This expression could be regarded as the 'object code' of the program for the λ-reduction machine that SIS provided. By applying this code to some input, and reducing again, one could get the output of the program according to the semantics.

The intended use of SIS was two-fold: 'debugging' semantic descriptions, by empirical testing of whether they gave the *intended* input-output behaviour for programs; and automatic generation of correct (prototype) implementations of programming languages from their semantic descriptions.

In the mid 1970's, denotational semantics was generally regarded as the most promising framework for semantic description. Adequate techniques had been developed for representing the semantics of all common (and many uncommon) constructs of programming languages. It was expected that before long, every major programming language would have a complete denotational description, which could then be given as input to SIS to provide (inefficient but) correct implementations. However, this expectation was not fulfilled—far from it.

It turned out that, despite the elegant and powerful theory of denotational semantics, there are severe pragmatic problems with applying it to languages of the scale of Pascal, C, etc. These problems can be observed already in the descriptions of small illustrative languages given in pedagogical texts on denotational semantics [40, 61, 64]: when changes or extensions to the described language require changes to the definitions of the semantic domains, the original semantic equations may need to be completely reformulated.

This is admittedly only a minor annoyance when dealing with small examples, but it becomes a serious hindrance when developing descriptions of larger languages: making extensions and changes to large denotational descriptions is simply too tedious and error-prone. It also prevents reuse of parts of a denotational description when describing a related language. Better modifiability and reusability are essential, especially if formal semantics is to be usable during the process of language design.

Another purely pragmatic problem is the difficulty of recovering fundamental concepts, such as order of execution or scopes for bindings, from their denotational description. The writer of the description may have a clear conceptual understanding of a programming language, but it gets obscured by the representation of the concepts directly in terms of higher-order functions on domains. In particular, implementors of programming languages are unlikely to refer to semantic descriptions unless the latter provide clear specification of the intended *operational* properties of language constructs.

The main causes of these pragmatic problems appear to be as follows:

1. The definitions of semantic domains are globally visible throughout a denotational description. (Denotational semantics was developed before the use of information-hiding modules and abstract data types became accepted practice for coping with problems of scale in software engineering.) Changes in domain definitions are often required when extending the described language with further constructs—e.g., a change from the direct style to the continuation style when adding jumps, or to power domains when adding nondeterminism. If we could anticipate all such changes, we could *start* with the more complex domains, but that would be unreasonable, as well as notationally burdensome.
2. The way λ-notation is used for specifying semantic entities depends strongly on the details of domain definitions. The classic example of this problem is provided by the semantics of statement sequences: with domains defined for the direct style of semantics, the denotations of the statements have to be composed normally; with the continuation style, the composition has essentially to be reversed. Also, the notation used for constructing elements of sum domains $D_1 + \cdots + D_n$ from the summands, and for case selection on such elements, is sensitive to the positions of the D_i in the sum: inserting a new summand in the middle can involve major changes throughout, just to preserve well-formedness of the notation.
3. All programming concepts have to be reduced to pure functions. This corresponds to translating arbitrary programs into a (lazy) functional programming language, and the amount of encoding required can be large. E.g., assigning values to variables is represented by composing functions that map stores to stores; the fact that the store remains 'single-threaded' is thoroughly obscured by the encoding.

What is needed to remedy the problems listed above?

To alleviate problem 1, one might think it would be sufficient to introduce explicit modular structure into denotational descriptions. This, however, doesn't help if all the module bodies still depend on the details of the domain definitions. It is essential for the modules to be specified in the style of abstract data types, providing notation for the required operations on elements of domains independently of their internal structure. Once this has been done, the explicit modular structure serves mainly as a *reminder* of the independence that has been obtained.

Regarding problem 2, one seems forced to introduce notation for *combining* denotations of program phrases without exploiting knowledge of their domain structure. Then only the definition of this notation, not its use, needs changing when the domains of denotations are changed. E.g., one might introduce notation for sequencing statement executions (as with monads [25, 27]), the use of this notation being independent of whether direct or continuation style domains are used. The pragmatic problems with positional notation for sums of domains could be addressed by using labelled sums (although there are still problems concerning sensitivity to nesting levels of summands).

Finally, to remedy problem 3, it appears necessary to introduce notation not just for sequencing but for all the *fundamental computational concepts* found in programming languages: scopes of bindings, storage of values, communication between concurrent processes, nondeterminism, etc. The use of such notation in semantic equations allows the conceptual analysis of constructs to be expressed directly, and its 'coding' in λ-notation is hidden in the definition of the notation.

1.2 Abstract Semantic Algebras

The above considerations of the pragmatic problems with denotational semantics led to the gradual development of the action semantics framework. At first [31] the idea was to keep as close as possible to denotational semantics, merely avoiding dependence on the structure of domains by introducing a few *combinators* for sequencing and data flow and defining them as auxiliary notation.

It soon became clear, however, that the combinators formed an interesting algebraic structure. The sequencing combinator was of course reminiscent of the composition of a category. Some of other combinators could be given familiar categorical interpretations, e.g., as source and target tupling; but it was not clear how best to accommodate further compositions, such as sequencing combined with dataflow, i.e., strict functional composition. So a pure categorical formulation was rejected in favour of a more general notion of *semantic algebra*—analogous to a data type, but with operations being combinators and primitives corresponding somehow to fundamental concepts of programming languages.

A series of papers on semantic algebras [33, 34, 35, 36] presented various sets of combinators, together with algebraic laws that they were supposed to obey, giving so-called *abstract* semantic algebras. The elements of abstract semantic algebras were always intended to have a clear operational interpretation; they were referred to as *actions* starting from around 1985 [11, 12, 49, 62, 70]. The combinators and primitives of the *action notation* have been rather stable since then—although the symbols used to denote them didn't stabilize until 1991 [42, 47, 63, 71].

1.3 Structural Operational Semantics

How should the intended interpretation of action notation be defined? Throughout the development of action semantics, much emphasis has been placed on the algebraic laws that are to be satisfied by the action combinators and primitives. It is however problematic to take such a set of laws as a definition: they might be inconsistent[3] or incomplete. Even if a consistent and complete set of laws could be found, it might well be difficult to see that they ensured the *intended* operational interpretation of action notation. Note that algebraic laws are also used in action semantics for specifying the data processed by actions, but there the problems of consistency and completeness are much less severe.

Instead, the current definition of action notation [42] is given using Structural Operational Semantics [60]. The required laws are then supposed to hold for a

[3] Inconsistent in the sense of having only trivial models.

derived testing equivalence (although in practice most of them can be verified using bisimulation).

Note that it would be difficult to provide a satisfactory denotational semantics for the full action notation: not only is it doubtful that a denotational model satisfying all the laws could be constructed using standard domains (one would need something close to a fully abstract model) but also action notation involves unbounded nondeterministic choice, which poses problems for ordinary continuity. To cite Abramsky (arguing for an intensional semantics based on games): "... once languages with features beyond the purely functional are considered, the appropriateness of modelling programs by functions is increasingly open to question. Neither concurrency nor 'advanced' imperative features have been captured denotationally in a fully convincing fashion." [1] Some of the action combinators and primitives are however quite easy to define as auxiliary notation in denotational semantics [25, 41].

It should be apparent from the above sketch of the development of action semantics that, formally, this framework is just a mixture of techniques from denotational and operational semantics, together with algebraic laws. *The only real originality of action semantics lies in the design of the action notation.*

In the next sections, we shall consider the concepts of action semantics more closely, give a simple illustration, and explain how the design of action notation ensures the desired pragmatic benefits.

2 Action Semantics

As indicated in Section 1, the starting point for the development of action semantics was denotational semantics. Action semantics has retained two of the main features of denotational semantics: the use of context-free grammars to define abstract-syntax trees; and the use of semantic equations to give inductive definitions of compositional semantic functions mapping such trees to semantic entities. (Action semantics may also be viewed as initial-algebra semantics [13].)

The essential deviation from conventional denotational semantics concerns the universe of semantic entities, and the notation used to specify individual entities.

Semantic entities are used to represent the implementation-independent behaviour of programs, as well as the contributions that parts of programs make to overall behaviour. There are actually three kinds of semantic entity used in action semantics: *actions*, *data*, and *yielders*. The main kind is actions; data and yielders are subsidiary. The notation used for specifying actions and the subsidiary semantic entities is called, unsurprisingly, *action notation*.

Actions are essentially dynamic, computational entities. The *performance* of an action directly represents information processing behaviour and reflects the gradual, step-wise nature of computation: each step of an action performance may access and/or change the *current information*. Yielders occurring in actions may access, but not change, the current information. The *evaluation* of a yielder always results in a data entity (including a special entity used to represent undefinedness). For example, a yielder might always evaluate to the datum currently stored in a particular cell, which could change during the performance of an action, and become undefined when the cell is freed.

2.1 Actions

A performance of an action, which may be part of an enclosing action, either: *completes*, corresponding to normal termination; or *escapes*, corresponding to exceptional termination; or *fails*, corresponding to abandoning an alternative; or *diverges*. Actions can be used to represent the semantics of programs: action performances correspond to possible program behaviours. Furthermore, actions can represent the (perhaps indirect) contribution that *parts* of programs, such as statements and expressions, make to the semantics of entire programs.

An action may be nondeterministic, having different possible performances for the same initial information. Nondeterminism represents implementation-dependence, where the behaviour of a program (or the contribution of a part of it) may vary between different implementations—or even between different instants of time on the same implementation.

The information processed by action performance may be classified as follows:

– *transient*: tuples of data, corresponding to intermediate results;
– *scoped*: bindings of tokens to data, corresponding to symbol tables;
– *stable*: data stored in cells, corresponding to the values assigned to variables;
– *permanent*: data communicated between distributed actions.

Transient information is made available to an action for immediate use. Scoped information, in contrast, may generally be referred to throughout an entire action, although it may also be hidden locally in a sub-action. Stable information can be changed, but not hidden, in the action, and it persists until explicitly destroyed. Permanent information cannot even be changed, merely augmented.

When an action is performed, transient information is given only on completion or escape, and scoped information is produced only on completion. In contrast, changes to stable information and extensions to permanent information are made *during* action performance.

The different kinds of information give rise to so-called *facets* of actions, focusing on the processing of at most one kind of information at a time:

– the *basic* facet, processing independently of information (control flows);
– the *functional* facet, processing transient information (actions are *given* and *give* data);
– the *declarative* facet, processing scoped information (actions *receive* and *produce* bindings);
– the *imperative* facet, processing stable information (actions *reserve* and *unreserve* cells of storage, and *change* the data stored in cells); and
– the *communicative* facet, processing permanent information (actions *send* messages, *receive* messages in buffers, and offer *contracts* to *agents*).

These facets of actions are independent. For instance, changing the data stored in a cell—or even unreserving the cell—does not affect bindings involving that cell.

The standard notation for specifying actions consists of *primitive actions* and action *combinators*. Each primitive action is single-faceted, affecting information in only one facet—although any yielders that it contains may refer to several kinds of information.

An action combinator determines control and information flow for each facet of the combined actions, allowing the expression of multi-faceted actions, such as an action that both (imperatively) reserves a cell of storage and then (functionally) gives the identity of the reserved cell. For instance, one combinator determines left-to-right sequencing together with left-to-right transient data flow, letting received bindings flow to its sub-actions; another combinator differs from that only regarding data flow: it concatenates any transients that the sub-actions give when completing, not passing transients between the actions at all. Some selections of control and information flow are disallowed, e.g., interleaving together with transient data flow between the interleaved sub-actions. In particular, the combination of imperative and communicative facets always follows the flow of control.

Note that actions with only a functional facet correspond quite closely to pure partial mathematical functions, the difference being that performance of a functional action may escape or fail, as well as completing or diverging.

2.2 Data

The information processed by actions consists of items of data, organized in structures that give access to the individual items. Data can include various familiar mathematical entities, such as truth values, numbers, characters, strings, lists, sets, and maps. It can also include entities with purely computational usage, such as tokens, cells, and agents—all used for accessing data from the current information—and some compound entities with data components, such as messages and contracts. Actions themselves are not data, but they can be incorporated in so-called *abstractions*, which are data, and subsequently *enacted* back into actions. (Abstraction and enaction are a special case of so-called *reification* and *reflection*.) New sorts of data can be introduced *ad hoc*, for representing special pieces of information.

2.3 Yielders

Yielders are entities that can be evaluated to yield data during action performance. The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the storage and message buffer. Evaluation cannot affect the current information. Compound yielders can be formed by the application of data operations to yielders.

2.4 Action Notation

The standard symbols used in action notation are ordinary English *words*. In fact action notation mimics natural language: terms standing for actions form imperative verb phrases involving conjunctions and adverbs, e.g., `check it and then escape`, whereas terms standing for data and yielders form noun phrases, e.g., `the items of the given list`. Definite and indefinite articles can be exploited for readability, e.g., `choose a cell then reserve the given cell` (formally, '`a`' and '`the`' denote the identity function).

These simple principles for choice of symbols provide a surprisingly grammatical fragment of English, allowing specifications of actions to be made fluently readable—without sacrificing formality at all. To specify grouping unambiguously, parentheses may be used.[4]

Compared to other formalisms, such as λ-notation, action notation may appear to lack conciseness: each symbol generally consists of several letters, rather than a single sign. But the comparison should also take into account that each action combinator usually corresponds to a complex pattern of applications and abstractions in λ-notation. For instance, (under the simplifying assumption of determinism) the action term '`A1 then A2`' might correspond to something like $\lambda\varepsilon_1.\lambda\rho.\lambda\kappa.A_1\varepsilon_1\rho(\lambda\varepsilon_2.A_2\varepsilon_2\rho\kappa)$. In any case, the increased length of each symbol seems to be far outweighed by its increased perspicuity.

For some applications, however, such as formal reasoning about program equivalence on the basis of their action semantics, optimal conciseness may be highly desirable, and it would be appropriate to use abbreviations for our verbose symbols. The choice of abbreviations is left to the discretion of the user. Such changes of symbols do not affect the *essence* of action notation, which lies in the standard primitives and combinators, rather than in the verboseness of the standard symbols.

The informal appearance and suggestive words of action notation should encourage programmers to read it, at first, rather casually, in the same way that they might read reference manuals. Having thus gained a broad impression of the intended actions, they may go on to read the specification more carefully, paying attention to the details. A more cryptic notation might discourage programmers from reading it altogether.

The intended interpretation of the standard notation for actions has been specified operationally, once and for all [42]. All that one has to do before using action notation is to specify the information that is to be processed by actions—it may vary significantly according to the programming language being described. This specification may involve *extending* data notation with further sorts of data, and *specializing* standard sorts, using sort equations. Furthermore, it may be convenient to introduce formal *abbreviations* for commonly-occurring, conceptually significant patterns of notation. Extensions, specializations, and abbreviations are all specified *algebraically*. The specification in Section 3 illustrates the use of sort equations to specialize some standard sorts of data, and to

[4] To avoid a plethora of parentheses in larger examples, typographic devices such as indentation and vertical lines may be used instead.

specify two nonstandard sorts of data for use in the semantic equations, namely `value` and `number`.

3 Illustrative Example

The example of an action-semantic description provided in this section is a very simple one. It serves merely as an illustration of the use of the main combinators of action notation. Some more interesting and realistic examples of ASDs are referenced in Section 7.

The example is divided into three modules, specifying abstract syntax, semantic functions, and semantic entities. The modules are written in the ASCII-format accepted by the ASD Tools [68], which were used to check their well-formedness.

```
module: Abstract Syntax.  grammar:

 (*)  Stmt  = [[Id ":=" Expr]]
            | [["if" Expr "then" Stmts "else" Stmts]]
            | [["while" Expr "do" Stmts]].

 (*)  Stmts = <Stmt <";" Stmt>*>.

 (*)  Expr  = Num | Id | [[Expr Op Expr]].

 (*)  Op    = "+" | "/=".

 (*)  Num   = [[digit+]].

 (*)  Id    = [[letter (letter|digit)*]].

endgrammar. closed. endmodule: Abstract Syntax.
```

Table 1. The SIMPL Illustrative Language

The grammar shown in Table 1 specifies several sorts of abstract-syntax trees, using a variant of BNF grammar allowing regular expressions. The details are not so important, but note that the double brackets `[[...]]` indicate node construction (in denotational semantics, they are used only to delimit syntactic phrases in semantic equations). The angle brackets `<...>` group components (thus `Stmts` is the sort of sequences of the form `<S1 ";"...";" Sn>`).

The semantic equations in Table 2 define semantic functions mapping abstract-syntax trees to semantic entities. Note the explicit specification of the dependence of this module on the other two (indicated by `needs:`).

```
module: Semantic Functions.  needs: Abstract Syntax, Semantic Entities.

introduces: execute_, evaluate_, the result of_.

variables:  I:Id; N:Num; E,E1,E2:Expr; O:Op; S:Stmt; S1,S2:Stmts.

 (*)  execute_ :: Stmts ->  action[completing|diverging|storing].

[1:]  execute [[I ":=" E]] =
        evaluate E then store the given number in the cell bound to I.

[2:]  execute [["if" E "then" S1 "else" S2]] =
        evaluate E then
        ( ( check the given truth-value and then execute S1 ) or
          ( check not the given truth-value and then execute S2 ) ).

[3:]  execute [["while" E "do" S1]] =
        unfolding
        ( evaluate E then
          ( ( check the given truth-value and then
              execute S1 and then unfold ) or
            ( check not the given truth-value ) ) ).

[4:]  execute <S ";" S2> = execute S and then execute S2.

 (*)  evaluate_ :: Expr -> action[giving a value].

[5:]  evaluate N = give decimal N.

[6:]  evaluate I = give the number bound to I or
                   give the number stored in the cell bound to I.

[7:]  evaluate [[E1 O E2]] =
        ( evaluate E1 and evaluate E2 ) then give the result of O.

 (*)  the result of_ :: Op -> yielder[of a value]
                                   [using given (value,value)].

[8:]  the result of "+" = the number yielded by
        the sum of (the given number#1, the given number#2).

[9:]  the result of "/=" = not (the given value#1 is the given value#2).

endmodule: Semantic Functions.
```

Table 2. SIMPL Action Semantics

The symbols introduced in ASDs may be prefix, postfix, infix, or more generally, 'mixfix'. There is a uniform precedence rule to allow omission of grouping parentheses: infixes have the weakest precedence (and associate to the left), then come prefixes, and finally postfixes. The place-holder '_' shows where the arguments go when operations are applied. Semantic functions, of course, take only one argument, and are conventionally denoted by prefix symbols.

The symbol `action` denotes the sort of all actions; a term `action[O]` denotes a subsort, including only those actions whose possible outcomes are contained in `O`. Similarly, a term of the form `yielder[of D][using I]` denotes the subsort of `yielder` that includes those yielders whose evaluation always returns a data item of sort `D`, referring at most to the current information indicated by `I`.

In equation 1, the functional action combination `A1 then A2` represents ordinary functional composition of `A1` and `A2`: the transients given by `A1` on completion are given only to `A2`. Regarding control flow, `A1 then A2` specifies normal left-to-right sequencing.

The yielder `given D` yields all the data given to its evaluation, provided that this is of the data sort `D`. For instance `the given number` (where '`the`' is optional) yields a single individual of sort `number`, if such is given. (Otherwise it yields the special entity `nothing`, which represents undefinedness, and similarly in other cases below.) The yielder `the D bound to T` refers to the current binding of sort `D` for the token `T`.

The primitive action `store Y1 in Y2` requires `Y1` to yield a storable value, and `Y2` to yield a cell.

In equation 2, the action `check Y` requires `Y` to yield a truth value; it completes when the value is true, otherwise it fails. The action `A1 or A2` represents implementation-dependent choice between alternative actions. If the alternative currently being performed fails, it is abandoned and, if possible, some other alternative is performed instead, i.e., *back-tracking*. Here, `A1` and `A2` are guarded by complementary checks, so the choice is deterministic.

The basic action combination `A1 and then A2` combines the actions `A1` and `A2` into a compound action that represents their normal, left-to-right sequencing, performing `A2` only when `A1` completes.

Equation 3 shows how iteration is specified in action semantics. The action combination `unfolding A` performs `A`, but whenever it reaches the dummy action `unfold`, it performs `A` instead.

Note that the semantics of a statement sequence is well-defined by equation 4, because the restriction on the sort of `S` ensures that the argument on the left-hand side of the equation can only match a statement sequence in one way.

In equation 5, the primitive action `give Y` completes, giving the data yielded by evaluating the yielder `Y`. The operation `decimal` is a standard data operation on strings.[5]

The two alternatives in equation 6 correspond to the cases that `I` is a constant or a variable identifier. The disjointness of the sorts `number` and `cell` ensures that

[5] An abstract-syntax tree whose direct components are all single characters is identified with the string of those characters.

the choice of an unfailing alternative is deterministic. When no binding for the identifier `I` to a number or to a cell is received, the action fails.[6]

The action `A1 and A2` used in equation 7 represents implementation-dependent order of performance of the indivisible sub-actions of `A1`, `A2`. When these sub-actions cannot 'interfere' with each other, as here, it indicates that their order of performance is simply irrelevant. Left-to-right order of evaluation can be specified by using the combinator `A1 and then A2` instead of `A1 and A2` above. In both cases, the values given by the sub-actions get tupled.

In equation 8, the yielder `the number yielded by Y` is used to insist that `Y` yields a value of sort `number` (the standard data operation `sum` might return an integer not in the `number` subsort).

The yielder `given Y#n` used in equations 8 and 9 yields the `n`'th individual component of a given tuple, provided that this component is of sort `Y`.

```
module: Semantic Entities.

includes: Action Notation.

introduces: value, number.

 (*)  token    =  string.
 (*)  bindable =  cell | number.
 (*)  storable =  number.
 (*)  value    =  number | truth-value.
 (*)  number   =< integer.

endmodule: Semantic Entities.
```

Table 3. Specializing Action Notation for SIMPL Semantics

The illustrative example of an ASD is completed by the module in Table 3, which specifies some sorts (`token`, `bindable`, `storable`) that are left open in the standard specification of `Action Notation`. (The use of `includes:` rather than `needs:` specifies that the imported notation is also exported.) The sorts `value` and `number` are introduced just for use in this illustrative ASD, and have no predetermined interpretation in `Action Notation`. Just as in Table 1, a vertical bar expresses union of sorts. The sort inclusion `number =< integer` leaves open whether `number` is bounded.

So much for the example, which should have given a rough impression of the 'look and feel' of action notation and action-semantic descriptions.

[6] As there are no declarations at all in SIMPL, the semantics of a SIMPL statement depends on received bindings for pre-declared identifiers.

4 Pragmatics

Let us now assess some of the pragmatic aspects of action-semantic descriptions. In particular, how well may we expect ASDs to scale up from illustrative examples (like the one given in the preceeding section) to practical programming languages?

In marked contrast to the situation with denotational-semantic descriptions, *making extensions and changes to an ASD generally affects only those parts of the description dealing directly with the constructs involved.*

E.g., adding expressions that allow function or process activation would *not* require any changes at all to the semantic equations given in Section 3. The enrichment of the actions representing expression evaluation by, e.g., the potential for side-effects or communication, does not invalidate the use of the combinator `A1 and A2` in equation 7, since it has a well-defined interpretation for actions with arbitrary information-processing capabilities (it interleaves the atomic steps).

This very desirable pragmatic aspect of action semantics depends on two crucial features of action notation:

– Each combinator is defined *universally* on actions. Contrast this with function composition in λ-notation, which requires exact matching of types between the composed functions.
– There is no mention of the presence or absence of any particular kind of information processing, except where creation or inspection of this information is required. For instance, stored information is referred to *only* in equations 1 and 6, whereas in a denotational semantics for the same language, every semantic equation would have to cater for the fact that all denotations are functions of the store.

It also depends on the fact that we may extend sorts of data (e.g., `bindable`) with new values or subsorts without disturbing the notation for creating or matching values. This is a feature of the algebraic specification framework used for the foundations of action semantics (summarized in the next section): it provides a genuine sort union operation, which behaves like set union, and avoids the pragmatic problems of the notation for sums of domains used in denotational semantics.

Since the above features ensure that ASDs have an *inherent* modularity, the use of explicit modules is almost redundant. In fact it is usual in ASDs to let all the semantic entities be visible throughout all the semantic equations, and this does not cause any problems with modifiability, etc.

Action semantics provides a high degree of extensibility, modifiability, and reusability, which is especially important when using semantic descriptions during language design, and highly significant when developing larger ASDs in general. An action-semantic description is also strongly suggestive of an operational understanding of the described language, as required by implementors. Moreover, it has been found to be very well suited for generating compilers [5, 56, 58] and interpreters [69].

Thus the pragmatic aspects of action semantics seem to be satisfactory.

5 Foundations

The foundations of action semantics are based on the framework of *unified algebras* [37, 38, 39]: each part of an ASD is interpreted as a unified algebraic specification.

The signature of a unified algebra is simply a ranked alphabet. The universe of a unified algebra is a (distributive) lattice with a bottom value, together with a distinguished subset of *individuals*. The operations of a unified algebra are required to be monotone (total) functions on the lattice; they are not required to be strict or additive, nor to preserve the property of individuality.

All the values of a unified algebra may be thought of as *sorts*, with the individuals corresponding to singleton sorts. The partial order of the lattice represents sort inclusion; join is sort union and meet is sort intersection. The bottom value (denoted by `nothing`) is a vacuous sort, often used to represent the lack of a result from applying an operation to unintended arguments. A special case of a unified algebra is a *power algebra*, whose universe is a power set, with the singletons as individuals.

The axioms of unified algebraic specifications are Horn clauses involving equations `T1=T2`, inclusions `T1=<T2`, and individual inclusions `T1:T2`. An equation holds when the terms have identical values, an inclusion holds when the values of the terms are in the partial order of the lattice, and an individual inclusion `T1:T2` holds when the value of `T1` is not only included in that of `T2`, but also in the distinguished subset of individuals.

Unified algebraic specifications always have initial models, because they are essentially just unsorted Horn clause logic (with equality) specifications, and the lattice structure and monotonicity of operations can all be captured by Horn clauses.

As illustrated in Section 3, an ASD consists of a grammar, some semantic equations, and a specification of the required universe of semantic entities. Thanks to the sort equations and inclusions allowed as axioms by unified algebras, all these parts have a straightforward interpretation, as follows.

Regarding grammars, each nonterminal symbol is interpreted as a sort constant, and the alternatives on the right-hand sides of productions as sort terms, combined with sort union; productions themselves are simply regarded as equational axioms [45], in contrast to the more elaborate interpretation of grammars as signatures usually taken in the literature [13].

The formal interpretation of a set of semantic equations is that the semantic functions are ordinary equationally-specified operations (taking a single syntactic argument and returning a semantic entity). The only use made here of the special features of unified algebras is in specifying subsorts of actions and yielders in the functionalities of the semantic functions. An alternative—but more complicated—interpretation would be to take account of the intended compositionality and inductiveness of the definitions, by regarding the semantic functions as the components of the unique homomorphism from the initial algebra of abstract syntax to a target algebra derived from the semantic equations.

The specification of the semantic entities consists of the specialization of action notation to particular sorts of data, together with the algebraic specification of abstract data types. The former involves sort equations and inclusions; the latter could be given in any decent algebraic specification framework. The operational semantics of the general action notation is fixed, and cannot be changed; it has been specified [42] in the style of structural operational semantics [60] (using an unorthodox presentation where the transition relation is a function from individual configurations to sorts, exploiting here unified algebras again). Notions of bisimulation and testing equivalence on actions are defined, completing the formalization of actions. The laws that action notation obeys are thus consequences of the definitions, rather than axioms. The next section surveys further work concerning the theory of actions.

6 Theory

After the experience with the development of denotational semantics, where much effort was spent on theoretical aspects, and the problems of applying the framework to practical languages were only realized after some time, I decided to proceed differently with action semantics: the first priority was to check that the pragmatic aspects of ASDs were satisfactory. Thus it is not surprising that a decent theory for action semantics has been slow to emerge. (Of course foundations of action semantics, such as those sketched in Section 5, were provided right from the start, otherwise it could not have been regarded as a formal framework at all.)

The theory of action semantics still has not been developed to the extent of, say, domain theory for denotational semantics. There may be many reasons for this: action notation may appear too large or unwieldy for theoretical analysis; or perhaps the notation conventionally used in ASDs is too verbose and informal-looking.

Nevertheless, significant work on several aspects of the theory of action semantics has already been done. The descriptions of it below are mostly adapted from the abstracts of the cited papers.

Note that the aim of this theoretical development is not only to investigate the properties of action notation *per se*, but also to allow reasoning about programs and programming languages by means of their action-semantic descriptions. For instance, algebraic laws established for action notation may be used to reason about program equivalence. (The situation is similar with regard to practical applications: flow analysis and code generation for action notation allow efficient compilation of programs according to their action semantics, as reported in Section 7.) The *possibility* of lifting analyses from action notation to programming languages depends on the compositionality of action semantics; its *usefulness* depends crucially on the simplicity of the actions that represent program constructs—and of course on the existence of action-semantic descriptions for programming languages of practical interest.

6.1 Type Inference

In papers in TCS and at the ESOP conference in 1990, Even and Schmidt [11, 12] formulated a model for action semantics based on Reynolds's *category-sorted algebra*. In the model, actions are natural transformations, and the composition operators are compositions in a 'category of actions'. They use the model to prove semantic soundness and completeness of a unification-based, decidable type-inference algorithm for action semantics expressions.

In a paper at ESOP'92, Doh and Schmidt [8] described a method that automatically extracts a type checking semantics, encoded as a set of type inference rules, from a category sorted algebra-based action-semantics definition of a programming language. The type inference rules are guaranteed to enforce strong typing, since they are based on an underlying meta-semantics for action semantics, which uses typing functions and natural transformations to give meaning. They use the type checking semantics to extract a dynamic semantics definition from the original action-semantics definition.

A distinguishing characteristic of action semantics is its facet system, which defines the variety on information flows in a language definition. The facet system can be analysed to validate the well-formedness of a language definition, and to calculate the operational semantics. At the Workshop on Action Semantics in 1994 [44] Doh and Schmidt [10] presented a single framework for doing all of this. The framework exploits the internal subsorting structure of the facets so that sort checking, static analysis, and operational semantics are related, sound instances of the same underlying analysis. The framework also suggests that action semantics' extensibility can be understood as a kind of 'weakening rule' in a 'logic' of actions. In the cited paper, the framework is used to perform type inference on specific programs, to justify meaning-preserving code transformations, and to 'stage' an ASD of a programming language into a static semantics stage and a dynamic semantics stage.

6.2 Provably-Correct Compiler Generation

As reported in his PhD thesis [58] and papers [57, 59] at ESOP and ICCL in 1992, Palsberg has designed, implemented, and proved the correctness of a compiler generator that accepts action-semantic descriptions of imperative programming languages. He has used it to generate compilers for both a toy language and a non-trivial subset of Ada. The generated compilers emit absolute code for an abstract RISC machine language that is assembled into code for the SPARC and the HP Precision Architecture. The generated code is an order of magnitude better than that produced by compilers generated by classical semantics-based compiler generators. His machine language needs no runtime type-checking and is thus more realistic than those considered in previous compiler proofs. He uses solely algebraic specifications; proofs are given in the initial model. The use of action semantics makes the processable language specification easy to read and pleasant to work with. His compiler generator may be seen as the first step towards user-friendly and automatic generation of realistic and provably correct compilers.

6.3 Action Analysis and Compiler Generation

The Actress system [5] accepts the action-semantic description of a source language, and from it generates a compiler. The generated compiler translates its source program to an action, performs sort inference on this action, (optionally) simplifies it by transformations, and finally translates it to object code. The sort inference provides valuable information for the subsequent transformation and code generation phases; Brown and Watt [4] reported their studies of the problem of sort inference for actions at the Workshop on Action Semantics. Transformations of the intermediate action greatly improve the efficiency of the object code; Moura's PhD thesis from 1993 [51] is concerned with these transformations, as reported at CC'94 [52].

Also at CC'94, Ørbæk [56] presented several analyses of actions. These allow his compiler generator (called OASIS) to generate efficient, optimizing compilers for procedural and functional languages with higher order recursive functions.

6.4 Action Equivalence

Lassen's PhD thesis work is devoted to developing a tractable theory for action equivalence. In a recent technical report [24] he has developed the foundations for a richer action theory, by bringing together concepts and techniques from process theory and from work on operational reasoning about functional programs. Semantic pre-orders and equivalences in the action semantics setting are studied and useful operational techniques for establishing contextual equivalences are presented. These techniques are applied to establish equational and inequational action laws and an induction rule for the basic facet of action notation.

Even more recently [23] Lassen has extended this theory to the functional and declarative facets, covering a substantial fragment of action notation involving transient and scoped information flow and higher-order, (unbounded) nondeterministic, and interleaving computation. Based on a reduction semantics for actions, operational reasoning techniques have been developed and used to establish an inequational theory for action notation. The potential of this theory is illustrated by proofs of various functional program equivalences via an action-semantic description of a functional language. He is currently extending the theory to the imperative facet in order to reason about practical, imperative programming languages.

7 Practice

The development of action semantics has so far involved rather modest resources, and there is still some way to go before a completely satisfactory set of examples and tools becomes available. The proceedings of the first International Workshop on Action Semantics [44] give a good impression of the level of activity in the area. This section lists some of the major applications so far.

7.1 ASDs of Programming Languages

Apart from the published ASDs mentioned below, various Master's theses at the University of Aarhus have covered large parts of various programming languages, e.g., ML, Amber, Joyce, Modula-3, and occam. Unfortunately, most of them were written in older versions of action notation, and are now out of print.

Pascal: The Pascal Action Semantics by Watt and the present author, available by FTP since 1993 [50], provides an almost complete formal specification of the dynamic semantics of Standard Pascal (Level 0). It is intended primarily as a 'showcase' example of the use of action semantics, rather than as a contribution to the understanding of Pascal. The current version is for readers who are already familiar with the action semantics framework. More work is needed before it is ready for a more general readership, including completion of an automated checking of the internal consistency and completeness of the specification, which is being done using the ASD Tools.

Standard ML: At the MFPS conference in 1988, Watt [70] reported on an ASD of the Standard ML 'bare' language in an early version of action notation. This has now been converted to use the current version of action notation, but not yet published. When ready—and extended to the whole of Standard ML—it will make an interesting basis for comparison between action semantics and structural operational semantics, as the latter approach was used for defining Standard ML [26].

The ANDF-FS: ANDF is an Architecture- and Language-Neutral Distribution Format developed by the Open Software Foundation (OSF) and other collaborators around the world. It is based on the TDF technology provided by the Defence Research Agency of the United Kingdom Ministry of Defence. It is a medium-level intermediate language, used as the target language of compilers for ordinary high-level languages. The production of the ANDF-FS, a formal specification of ANDF, was one of the tasks of the ESPRIT project OMI/GLUE [15].

In a technical report from 1993 [66], Toft assessed the feasibility of using various frameworks for the ANDF-FS: a hybrid of denotational and algebraic semantics; structural operational semantics, expressed as an algebraic specification in RSL; and action semantics, where the definition of action notation may either be expressed as higher-order functions or in terms of an operational semantics. His main conclusion was that "action semantics with an underlying structural operational semantics is the most qualified candidate for the ANDF-FS".

Unfortunately, at the time there was no mature tool support for action semantics. A compromise was found: the RAISE Specification Language RSL [65] is a general-purpose specification language with good tool support; by giving a structural operational semantics of the required subset of action notation in RSL, the RSL tools could be used for action semantics.

The ANDF-FS from 1994 [55], presented also at the Workshop on Action Semantics [16], was the first example of 'industrial' interest in action semantics.

Somewhat surprisingly, action semantics was used not only for the dynamic semantics of ANDF but also for its *static* semantics, exploiting action combinators to express static evaluation, type-checking, scopes of bindings, etc.

Particularly encouraging is the fact that the ANDF-FS wasn't put on the shelf to gather dust after completion. It was used by a group at the OSF Research Institute in Grenoble, for three purposes: to discuss refinement of the informal ANDF specifications, as an aid to develop a validation suite for ANDF, and as an aid to develop an interpreter. Despite little previous exposure to formal semantics, this group confirms that they found the ANDF-FS quite accessible—thanks at least partly to the verbosity of action notation.

Another significant aspect is that the project was conducted by persons who had not been involved in the development of action semantics. Even though the project took place in Denmark, my contact with it was limited to answering one question about my book.

7.2 Concurrent Languages

An ASD for a non-trivial sublanguage of Ada (including tasks) was given by the present author in his text book on action semantics [42]. He also reported the use of action semantics to describe concurrent languages in a REX workshop paper in 1992 [43]. His paper together with Krishnan at RTFT'92 [21] on specifying asynchronous transfer of control led to a further paper by Krishnan [19].

Musicante reported on an ASD for the Sun RPC protocol in 1992 [53]. With the present author he investigated a minor potential extension of the communicative facet of action notation to provide shared storage [54]. Together they presented an ASD of a small fragment of Standard ML at the FME'94 [48], and showed that adding concurrency primitives to the described language requires only extensions, not changes, to the description of the sequential constructs.

7.3 Tools

The ASD Tools have been developed since 1993 by van Deursen and the present author, and demonstrated at various conferences, e.g., AMAST'96 [68]. They provide a support environment for using action semantics, including facilities for parsing, syntax-directed (and textual) editing, checking, and interpretation of action-semantic descriptions. Such facilities significantly enhance accuracy and productivity when writing and maintaining large specifications, and are also particularly useful for those learning how to write ASDs. The notation supported by the ASD Tools, illustrated in Section 3, is a direct ASCII representation of the standard notation used for ASDs in the literature.

The ASD Tools are implemented using the ASF+SDF system [18], which is itself based on the Centaur system (developed by INRIA, among others). A licence for Centaur is required; this is available free of charge to academic institutions, and the entire ASD Tools implementation can be obtained by FTP. The ASD Tools were taken as a case study in the use of ASF+SDF by van Deursen in his PhD thesis [67].

The Actress system developed by Watt's group at Glasgow provides prototype tools, reported at CC'92 [5] and CC'94 [52], for interpreting action notation and for compiling it into C. The interpreter deals with most of the standard action notation except for the communicative facet.

In his PhD thesis [6] in 1992, and later in a joint paper with Schmidt [9], Doh presented a methodology for compiler synthesis based on action semantics. Each symbol of action notation is assigned specific 'analysis functions', such as a typing function and a binding-time function. When a language is given an action semantics, the typing and binding-time functions for the individual actions compose into typing and binding-time analyses for the language; these are implemented as the type checker and static semantics processor, respectively, in the synthesized compiler. Other analyses can be similarly formalized and implemented.

Partial evaluation has been used extensively in connection with compiler generation. In 1993, Bondorf and Palsberg [3] used the Similix system to obtain an action compiler by partial evaluation of an action interpreter. In a paper at PEPM'95, Doh proposed using partial evaluation for action transformation [7].

The compiler generator OASIS, presented by Ørbæk at CC'94 [56], is capable of generating efficient, optimizing compilers for procedural and functional languages with higher-order recursive functions. The automatically generated compilers produce code that is comparable with code produced by handwritten compilers. This work is perhaps the most convincing evidence so far of the practical applicability of action-semantics-based compiler generation.

8 Prospects

We have reviewed the current state of theory and practice of action semantics. What prospects are there for the future development and use of this framework?

Regarding the theory of action semantics: despite recent progress with proof techniques for action equivalence, much remains to be done in that direction. In particular, it is urgent to develop a useful proof calculus for the communicative facet of actions, which is based on *asynchronous* processes and message-passing; the work of Agha, Mason, Smith, and Talcott [2] appears to be applicable here.

The 'official' notation for actions and data has remained the same for the past five years. Some potential improvements have recently been suggested [22], and this may be a good time to make a revised version, taking into account the experience gained through using the present version in large ASDs. One of the major issues here is whether it might be better to take 'yielders' merely as a subsort of actions, their evaluation then being a special case of action performance. More superficially, different vocabularies of symbols for action combinators and primitives might be developed—perhaps even using graphics for visualization [20].

It is unclear whether users of action semantics should be allowed to change the operational semantics of action notation, or add new action combinators and primitives, e.g. as required for letting agents share storage [54]. Such changes would require re-proving all the laws of action notation. In any case, the current structural operational semantics of action notation is not so easy to modify; alternative forms of operational semantics, such as evolving-algebra semantics [14], might be preferable in that respect.

The underlying algebraic specification language used in action semantics is currently the somewhat unorthodox and lesser-known framework of 'unified algebras', summarized in Section 5. It might be advisable to replace it by a framework with a more direct set-theoretic basis [17] or by a sublanguage of the common algebraic specification language that is currently being developed by the Common Framework Initiative [46]. In either case, algebraic data-type definitions should be included, as this would allow a significant reduction in the size of the modules that specify auxiliary semantic entities.

To further encourage the practical use of action semantics, an integrated tool set supporting the writing, editing, and testing of ASDs should be provided. In particular, navigating and searching in larger ASDs needs to be made easier than with the existing tools, e.g., by using hyper-text links and indexing.

Then there is the matter of completing the existing partial ASDs of a number of practical programming languages, and of developing new ones, such as for the Java language. The larger ASDs that have already been produced confirm that action semantics has (much) better applicability than denotational semantics, so it should be feasible to build up an on-line library of checked ASDs in the near future. Large parts of these ASDs could then be reused in the descriptions of further languages. Moreover, they would provide appropriate input for compiler generators based on action semantics, avoiding a recurrence of the problems experienced with exploiting SIS for denotational semantics some 20 years ago, and perhaps stimulating further research and development of semantics-based compiler generation.

Finally, it is important for the future of action semantics that it gets taught in semantics courses at the undergraduate level. For this purpose, it may be best to provide a cut-down version of action notation, including only those symbols needed for describing familiar languages such as Pascal and Standard ML; the operational semantics of such a subset would probably be significantly simpler than that defining the full notation.

Up-to-date information about action semantics may be found via the Action Semantics Home Page, URL: `http://www.brics.dk/Projects/AS`.

Acknowledgements

I would like to take this opportunity to thank all those who have contributed so far to the development of action semantics, and to colleagues at the University of Aarhus and elsewhere for their encouragement to continue with this work. Thanks especially to Olivier Danvy, Christian Fabre, Søren B. Lassen, Peter Ørbæk, and David A. Watt, who all suggested significant improvements to earlier versions of this paper.

I would also like to thank the organizers of MFCS'96 for inviting me to present this paper, and for letting me have some extra pages.

My own part in the development of action semantics has been supported by: the Department of Computer Science, University of Aarhus; the Danish National Research Foundation Centre BRICS; the Danish Science Research Council project DART (5.21.08.03); and ESPRIT Basic Research Working Group COMPASS (3264 and 6112).

References

1. S. Abramsky. Semantics of interaction. In *Trees in Algebra and Programming – CAAP'96, Proc. 21st Int. Coll., Linköping*, volume 1059 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 1996.
2. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. To appear in *Journal of Functional Programming*, 1994.
3. A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *FPCA'93, Proc. Sixth ACM Conf. on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 308–317, 1993.
4. D. Brown and D. A. Watt. Sort inference in the Actress compiler generator. In [44], pages 81–98, 1994.
5. D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer-Verlag, 1992.
6. K.-G. Doh. *Action Semantics-Directed Prototyping.* PhD thesis, Kansas State University, 1992.
7. K.-G. Doh. Action transformation by partial evaluation. In *PEPM'95, Proc. ACM/SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Transformation, La Jolla, California*, pages 230–240, 1995.
8. K.-G. Doh and D. A. Schmidt. Extraction of strong typing laws from action semantics definitions. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1992.
9. K.-G. Doh and D. A. Schmidt. Action semantics-directed prototyping. *Comput. Lang.*, 19(4):213–233, 1993.
10. K.-G. Doh and D. A. Schmidt. The facets of action semantics: Some principles and applications (extended abstract). In [44], pages 1–15, 1994.
11. S. Even and D. A. Schmidt. Category sorted algebra-based action semantics. *Theoretical Comput. Sci.*, 77:73–96, 1990.
12. S. Even and D. A. Schmidt. Type inference for action semantics. In *ESOP'90, Proc. European Symposium on Programming, Copenhagen*, volume 432 of *Lecture Notes in Computer Science*, pages 118–133. Springer-Verlag, 1990.
13. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, 1977.
14. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
15. B. S. Hansen and J. Bundgaard. The role of the ANDF formal specification. Technical Report 202104/RPT/5, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1992.
16. B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In [44], pages 34–42, 1994.

17. C. Hintermeier, H. Kirchner, and P. D. Mosses. Combining algebraic and set-theoretic specification. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, 1995, Selected Papers*, volume 1130 of *Lecture Notes in Computer Science*, pages 255–274. Springer-Verlag, 1996. To appear.
18. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
19. P. Krishnan. Specification of systems with interrupts. *J. Systems Software*, 21:291–304, 1993.
20. P. Krishnan, B. McKenzie, and S. Hunt. Guile: Graphical user interface for linguistic experiments. In *Proceedings of the 17th Annual Computer Science Conference*, Australian Communications, pages 309–320, 1994.
21. P. Krishnan and P. D. Mosses. Specifying asynchronous transfer of control. In *RTFT'92, Proc. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, Delft*, volume 571 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
22. S. B. Lassen. Design and semantics of action notation. In [44], pages 34–42, 1994.
23. S. B. Lassen. Action semantics reasoning about functional programs. BRICS, Dept. of Computer Science, Univ. of Aarhus, Dec. 1995.
24. S. B. Lassen. Basic action theory. Technical Report RS-95-25, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1995.
25. S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.
26. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
27. E. Moggi. Computational lambda-calculus and monads. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 14–23. IEEE, 1989.
28. P. D. Mosses. The mathematical semantics of Algol60. Tech. Mono. PRG–12, Programming Research Group, Univ. of Oxford, 1974.
29. P. D. Mosses. *Mathematical Semantics and Compiler Generation*. D.Phil. dissertation, University of Oxford, 1975.
30. P. D. Mosses. Compiler generation using denotational semantics. In *MFCS'76, Proc. Symp. on Math. Foundations of Computer Science, Gdańsk*, volume 45 of *Lecture Notes in Computer Science*. Springer-Verlag, 1976.
31. P. D. Mosses. Making denotational semantics less concrete. In *Proc. Int. Workshop on Semantics of Programming Languages, Bad Honnef*, pages 102–109. Abteilung Informatik, Universität Dortmund, 1977. Bericht nr. 41.
32. P. D. Mosses. SIS, Semantics Implementation System: Reference manual and user guide. Tech. Mono. MD–30, Dept. of Computer Science, Univ. of Aarhus, 1979. Out of print.
33. P. D. Mosses. A constructive approach to compiler correctness. In *ICALP'80, Proc. Int. Coll. on Automata, Languages, and Programming, Noordwijkerhout*, volume 85 of *Lecture Notes in Computer Science*, pages 449–469. Springer-Verlag, 1980.
34. P. D. Mosses. A semantic algebra for binding constructs. In *Proc. Int. Coll. on Formalization of Programming Concepts, Peñiscola*, volume 107 of *Lecture Notes in Computer Science*, pages 408–418. Springer-Verlag, 1981.

35. P. D. Mosses. Abstract semantic algebras! In *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*, pages 45–71. North-Holland, 1983.

36. P. D. Mosses. A basic abstract semantic algebra. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis*, volume 173 of *Lecture Notes in Computer Science*, pages 87–107. Springer-Verlag, 1984.

37. P. D. Mosses. Unified algebras and action semantics. In *STACS'89, Proc. Symp. on Theoretical Aspects of Computer Science, Paderborn*, volume 349 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

38. P. D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989.

39. P. D. Mosses. Unified algebras and modules. In *POPL'89, Proc. 16th Ann. ACM Symp. on Principles of Programming Languages*, pages 329–343. ACM, 1989.

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

41. P. D. Mosses. A practical introduction to denotational semantics. In *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 1–49. Springer-Verlag, 1991.

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

43. P. D. Mosses. On the action semantics of concurrent programming languages. In *Semantics: Foundations and Applications, Proc. REX Workshop, Beekbergen, 1992*, volume 666 of *Lecture Notes in Computer Science*, pages 398–424. Springer-Verlag, 1993.

44. P. D. Mosses, editor. *Proc. 1st Intl. Workshop on Action Semantics, Edinburgh, 1994*, number NS-94-1 in BRICS Notes Series. BRICS, Dept. of Computer Science, Univ. of Aarhus, 1994.

45. P. D. Mosses. Unified algebras and abstract syntax. In *Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, 1992, Selected Papers*, volume 785 of *Lecture Notes in Computer Science*, pages 280–294. Springer-Verlag, 1994.

46. P. D. Mosses, editor. *CoFI: Initiative for a Common Framework for Algebraic Specification*, URL: http://www.brics.dk/Projects/CoFI, 1996.

47. P. D. Mosses. A tutorial on action semantics. 50pp. Tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996), Mar. 1996.

48. P. D. Mosses and M. A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *Lecture Notes in Computer Science*, pages 461–479. Springer-Verlag, 1994.

49. P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.

50. P. D. Mosses and D. A. Watt. Pascal action semantics, version 0.6. URL: ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/MossesWatt93DRAFT.ps.Z, Mar. 1993.

51. H. Moura. *Action Notation Transformations.* PhD thesis, Dept. of Computing Science, Univ. of Glasgow, 1993.

52. H. Moura and D. A. Watt. Action transformations in the Actress compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1994.
53. M. A. Musicante. The Sun RPC language semantics. In *Proceedings of PANEL'92, XVIII Latin-American Conference on Informatics*. Universidad de Las Palmas de Gran Canaria, 1992.
54. M. A. Musicante and P. D. Mosses. Communicative action notation with shared storage. Tech. Mono. PB–452, Dept. of Computer Science, Univ. of Aarhus, 1993.
55. 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.
56. P. Ørbæk. OASIS: An optimizing action-based compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1994.
57. J. Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *ICCL'92, Proc. Fourth IEEE Int. Conf. on Computer Languages, Oakland*, pages 117–126. IEEE, 1992.
58. J. Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Dept. of Computer Science, Univ. of Aarhus, 1992. xii+224 pages.
59. J. Palsberg. A provably correct compiler generator. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434. Springer-Verlag, 1992.
60. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
61. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.
62. D. A. Schmidt. *The Structure of Typed Programming Languages*. The MIT Press, 1994.
63. K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.
64. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
65. The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice-Hall, 1992.
66. J. U. Toft. Feasibility of using RSL as the specification language for the ANDF formal specification. Technical Report 202104/RPT/12, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1993.
67. A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, Univ. of Amsterdam, 1994.
68. A. van Deursen and P. D. Mosses. ASD: The action semantic description tools. In *AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology, Munich*, volume 1101 of *Lecture Notes in Computer Science*, pages 579–582. Springer-Verlag, 1996.
69. D. A. Watt. Executable semantic descriptions. *Software – Practice and Experience*, 16:13–43, 1986.
70. D. A. Watt. An action semantics of Standard ML. In *Proc. Third Workshop on Math. Foundations of Programming Language Semantics, New Orleans*, volume 298 of *Lecture Notes in Computer Science*, pages 572–598. Springer-Verlag, 1988.
71. D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

Addendum: This paper also appears in *MFCS'96, Proc. 21st International Symposium on Mathematical Foundations of Computer Science, Cracow, Poland, September 1996*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996.