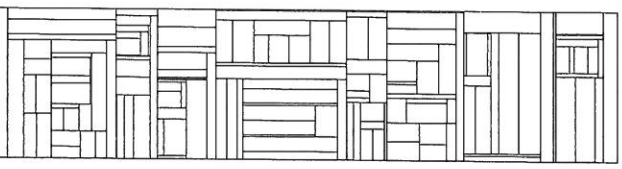
ISSN 0105-8517

The Potential Use of Action Semantics in Standards

Peter D. Mosses
David A. Watt

DAIMI PB – 206
April 1986

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 71 88 *Telex: 64767 aausci dk*



Abstract

Current standards for programming languages generally use informal, rather than formal, semantic descriptions. Possible reasons for this are discussed. Action Semantics, which has been developed from Denotational Semantics and Abstract Semantic Algebras, has some features that may make it more attractive for use in standards than other formal approaches. This paper describes and motivates Action Semantics, and gives some realistic examples of its use.

THE POTENTIAL USE OF ACTION SEMANTICS IN STANDARDS

(Position Paper for ISO ad hoc Group on Formal Semantics)

Peter D. Mosses David A. Watt*

1 Introduction

For many years now, BNF (with minor variations) has been used for describing the context-free syntax of programming languages. It has been found to be generally acceptable and useful - to language designers and standardizers, to programmers and implementors, and in formal studies.

In contrast, there is no such consensus with semantic descriptions. There is a diversity of approaches, each having advantages for some applications, but also having disadvantages for others. In particular, none of the current approaches to formal semantics seems to be appropriate for use in standards for programming languages, and standardizers have generally kept to informal, natural-language descriptions of semantics [2,3]. But it seems doubtful that informal descriptions of such complex artefacts as programming languages can ever be made sufficiently precise to rule out mis-interpretation by implementors and programmers; likewise, they cannot be used reliably in connection with program verification.

The main approaches to formal semantics are Denotational Semantics, Axiomatic Semantics and Operational Semantics. Let us look at them briefly in turn, and consider their strengths and weaknesses.

Denotational Semantics [21,23,22,12] is perhaps the main contender for use in standards. It seems that it is able to cope with everything that language designers can come up with, and it is founded upon an elegant and powerful theory. But there are problems with the comprehensibility of denotational descriptions: the lambda-notation used in the semantic equations is a foreign language to most programmers and implementors, requiring a considerable investment of time and effort to

---

*Computing Science Department, The University, Glasgow, Scotland G12 8QQ

master; and descriptions of full-scale languages are extremely difficult to grasp, due to poor modularity.

There have been some attempts to "sugar" the notation used in Denotational Semantics. In particular, the meta-language of VDM (usually referred to as "Meta-IV") [7] provides notation for a variety of mathematical constructs, such as sets and mappings, and has imperative features. The foundations of VDM and its precise relationship to lambda-notation have never been given in full. To deal with concurrency and non-determinism, Meta-IV would need to be extended [10].

The (largely) functional programming language ML [15] is comparable to Meta-IV as a meta-language for expressing denotational descriptions: imperative features are available, and ML would need extending to deal with concurrency and non-determinism. The type abstraction facility in ML could probably be used to provide the various mathematical data types that are built into Meta-IV; it could also be used to provide modularity. Unfortunately, there is as yet no formal definition of Standard ML. The ML implementation, however, can be used for the empirical testing of semantic descriptions (and for rapid prototyping [24]).

It is even possible to use a general-purpose programming language as a meta-language for expressing denotational descriptions: variants of Pascal, Algol68, C, and Ada have all been proposed for this purpose. This might seem attractive, due to the use of familiar notation, and the possibility of achieving modularity. However, general-purpose programming languages, even when reduced to an applicative subset and extended to allow higher-order functions (which makes them less familiar!), turn out to be rather clumsy for expressing denotational descriptions [1]. They also lack formal definitions themselves, making it impossible to reason about the semantics given to programs in the described language (especially with meta-circular descriptions!). Finally, it is not at all clear that it is desirable to reduce the semantics of new languages to the (often idiosyncratic) semantics of existing languages.

Axiomatic Semantics [13,4] is reasonably accessible to programmers and implementors, being based directly on axioms and inference rules. The main problems here are with generality and comprehensibility - some simple programming constructs, like procedures with parameters, have uncomfortably intricate descriptions - and with ensuring consistency. The published axiomatic description of Pascal [14] does not cover the full language, and, moreover, the rules for function declarations give logical inconsistency [5].

By the way, the so-called "weakest precondition" semantics of [9] is

not really axiomatic, even though it uses assertions: it is denotational, and suffers from much the same drawbacks as the usual denotational approach.

Operational Semantics [25] is just abstract programming of compilers and interpreters, and as such is quite easy for programmers and implementors to work with. However, operational descriptions of full-scale languages are rather voluminous documents, and little attention has been paid to modularity. There is a constant danger of operational specifications being biased towards particular implementation strategies, making it difficult to relate them to (real) implementations based on alternative strategies. Even when there is no bias towards a particular implementation strategy, there are certain to be details in the operational description that are of an implementation nature and not essential to an abstract understanding of the programming language's semantics. It is presumably this that accounts for the voluminousness.

The Structural Operational Semantics approach [20,19,8] essentially gives an axiomatic description of operational transitions. It can be used for static semantics and translation, as well as dynamic semantics. It has yet to be tried out "in the large", although a modified version, SMoLCS [6], is currently being used in an attempt to give a formal description of Ada. Both Structural Operational Semantics and SMoLCS seem to have some advantages over the pure axiomatic and operational approaches.

A weakness shared by all the above approaches is the lack of any explicit relation to familiar computational concepts, such as order of computation, scope rules, etc. The reader of a formal semantic description is forced to rediscover the concepts that were in the mind of the language designer in the first place. Informal semantic descriptions, when well-written, can avoid this problem, and possibly this is the major factor that causes programmers to prefer them to formal descriptions.

We are proposing a new approach to semantic description, one that attempts to avoid the disadvantages of the approaches mentioned above. Our approach is still evolving, and we are currently conducting a large-scale experiment - a full semantic description of ISO Standard Pascal (Level 0) - to show how well we can cope with conventional sequential languages. This experiment is nearing completion, and the results so far look promising [18]. But much work is needed to show that we can also handle concurrent and non-conventional languages. Thus we are not (yet) in a position to propose our approach for adoption in standards documents.

The new approach is called Action Semantics. It is a development of "Abstract Semantic Algebras" [16,17], but the emphasis is now on the user interface, rather than on the underlying foundations.

In essence, Action Semantics is denotational (or "compositional"), in that the semantics of each phrase is expressed in terms of the semantics of its subphrases. The denotations, however, are no longer taken to be higher-order functions, expressed in lambda-notation: they are "actions", which have a (reasonably) simple operational interpretation - and quite nice algebraic properties.

We shall see some examples of actions later. The semantics of a wide class of programming languages (both functional and imperative) can, it seems, be given in terms of a fairly small number of standard primitive actions and action combinators. This gives the possibility of re-using parts of previous semantic descriptions when describing new languages, and facilitates the semantic comparison of languages. Moreover, actions enjoy a high degree of orthogonality, which gives good modifiability of semantic descriptions.

The proposed notation for actions is intentionally verbose and suggestive. This, we believe, makes it possible to gain a (broad) impression of a language's semantics from a casual reading of its action-semantic description - and may encourage the casual reader to become a serious reader! There seem to be substantial pragmatic advantages, both for the readers and the authors of a semantic description, in having a notation with a fair amount of redundancy. Our action notation mimics natural language, but remains completely formal. Of course other, more "mathematical", representations of the notation could be used when conciseness is a primary concern.

Action Semantics was originally developed for use in specifying just the dynamic semantics of programming languages. Recently, we have become attracted by the idea of using it for expressing the checking of static constraints as well. Both the static and dynamic semantics can thus be specified as mappings from context-free abstract syntax to actions, using the same notation.

Our concern for the casual reader is reflected in the organization of action-semantic descriptions. First, an action semantics introduces the standard action notation to be used, giving an informal explanation of how actions can be "performed". The formal specification of actions themselves is relegated to an appendix, and may be ignored by the casual reader.

Then comes an abstract syntax for the programming language, and its

relation to concrete syntax. The abstract syntax is context-free, and is assumed both by the static semantics and by the dynamic semantics, which are given separately. (We prefer to avoid the introduction of an intermediate abstract syntax that incorporates statically-determinable context-sensitive information, even though that might permit a more concise dynamic semantics.)

The static and dynamic semantics both have the same form: they start by introducing the necessary value sorts and operations, followed by ad hoc abbreviations for commonly-occurring patterns of standard actions; then come the semantic equations, expressing the semantics of each phrase of abstract syntax in terms of the semantics of its subphrases, using the action notation.

If the formal specification of actions is ignored, we are left with what could be called a "formalizable" semantics. We suggest that such partial formal descriptions might be acceptable to programmers and implementors as an alternative to informal semantic descriptions, with the denotational structure and the standard notation for actions being regarded as merely a useful discipline for organizing informal-looking descriptions.

The formal specifications of standard actions that we prefer to give are algebraic. This facilitates the introduction of new actions (as may be needed for expressing new computational concepts); and algebraic axioms provide useful information about the properties of actions. Note that we do not expect the reader to acquire an intuitive grasp of actions just by gazing at the axioms for them; rather, the axioms may be used to "fine-tune" a previously-established conceptual understanding.

We shall not dwell on the algebraic specification of actions in this paper. We refer the reader to [17] for examples and a discussion of foundational aspects.

In the rest of the paper, we shall first indicate the meta-notation used in Action Semantics, and explain the concept of actions. Then we shall give some examples of the use of Action Semantics, taken from the current version of our Pascal semantic description.

2 Action Semantics

The meta-notation used in Action Semantics consists of BNF (for abstract syntax), parts of OBJ2 [11] (for specifying actions and values) and semantic equations (for specifying semantic functions). It is fully formal.

Now for the concept of actions. An action is just an entity that can be performed. The outcome of performing an action may be either completion, escape, or non-termination, or else the performance of the action might fail (i.e., have no outcome). An action receives information of various kinds: immediate values; values contained in variables; values bound to identifiers; input and output. An action may likewise produce information of the same kinds (provided that it completes or escapes).

Our standard actions are chosen to correspond to familiar computational notions. (The following informal descriptions of actions are very brief. They are intended only to give a rough impression of what the actions mean.)

An action may be primitive or compound. A primitive action may contain terms (T) that refer to information received by the action, and may introduce a name (N) for the value (V) of a term. Here are some examples of standard primitive actions:

- 'obtain an N from T' simply gives the name N to the value of T (but fails if the value is not of the sort indicated by N).
- 'check T is V' fails unless the value of T is V.
- 'create an N' creates a simple variable, and gives the name N to (the identity of) the variable.
- 'store T1 in T2' updates the contents of the variable (whose identity is) given by T2, to the value of T1.
- 'bind T1 to T2' binds the identifier given by T1 to the value of T2.
- 'skip' is a dummy action.

There are standard action combinators for expressing fundamental ways of combining actions. For example, if A1 and A2 are actions, then so are the following (symbols in square brackets below are optional):

- '[either] A1 or A2' chooses one of A1 or A2, but if the chosen action fails the other one is chosen instead.

- '[both] A1 and A2' performs A1 and A2 in an implementation- dependent order.
- '[first] A1 then A2' performs A1 and A2 sequentially.
- 'A1 before A2' accumulates bindings produced by A1 and A2.
- 'A1 where # = A2' is iteration; A1 is performed, with unfolding of A2 at occurrences of the dummy action '#'.
- 'enact T' performs the action encapsulated in the abstraction given by the value of T.
- 'block A' performs A, but restricts the scope of any bindings produced by A.

It is sometimes convenient to introduce ad-hoc abbreviations for commonly-occurring patterns of standard actions. For instance, in the Pascal Action Semantics, we introduce:

- 'coerce an N from T', which encapsulates all the implicit coercions in Pascal (integer to real, variable to current value, etc.);
- 'allocate an N of T', which extends 'create' to compound variables with type given by T; and
- 'assign T1 to T2', which extends 'store' to compound variables.

Different performances of an action may have different outcomes, corresponding to non-determinism or implementation-dependence. We say that an action A' is a correct implementation of another action A when (in any context) the outcome of each performance of A' is the outcome of some performance of A. This can be reformulated to give correctness criteria for real implementations of programming languages, based on their action semantics.

3 Examples of Use

In this section, we shall illustrate the use of Action Semantics with (somewhat simplified) excerpts from the semantic description of Standard Pascal that we are currently developing [18]. First, some semantic equations for the dynamic semantics. (Note that '!' starts an informal, end-of-line comment; the rest is completely formal!)

```
evaluate: Expression -> Action
!     'evaluate E' computes the un-coerced value of E.
!     It produces a result.
!     It may make changes to variables.
!     It produces no bindings.

evaluate [ I ] =
      obtain a result from binding-of(id I)

evaluate [ E1 DO E2 ] =
      both  evaluate E1 then
            coerce a 1st operand from the result
      and   evaluate E2 then
            coerce a 2nd operand from the result
      then  operate DO then
            obtain a result from the operator-result

...

operate: Dyadic-Operator -> Action
!     'operate DO' applies DO to two operands.
!     It receives a 1st operand and a 2nd operand,
!     and it produces an operator-result.
!     It makes no changes to variables.
!     It produces no bindings.

...
```

```
execute: Statement -> Action
!     'execute S' executes S.
!     It produces no values.
!     It may make changes to variables.
!     It produces no bindings.

execute [ S1 ";" S2 ] =
      first execute S1
      then  execute S2

execute [ "while" E "do" S ] =
      # where # =
            first evaluate E
            then  coerce a boolean from the result
            then  either
                        check the boolean is true then
                        execute S then #
                  or
                        check the boolean is false then
                        skip

execute [ VD ";" "begin" S "end" ] =
      block establish VD
            before execute S

...

establish: Variable-Declarations -> Action
!     'establish VD' executes the declarations in VD.
!     It produces no values.
!     It creates variables.
!     It produces bindings.

establish [ "var" I ":" TY ] =
      first typify TY
      then  allocate a variable of the type
      then  bind id I to the variable

...
```

The static semantic action for a program phrase just fails unless the program phrase satisfies the static constraints. Note that the action notation used in the static semantic equations (below) is essentially a restriction of the notation used in the dynamic semantics - albeit over different sorts of values. In particular, there is no need for creating variables, nor for potentially non-terminating iterations in static semantics.

```
evaluate-mode: Expression -> Action
!     'evaluate-mode E' checks E and deduces its mode.
!     It produces a result-mode.
!     It produces no bindings.

evaluate-mode [ I ] =
      obtain a result-mode from binding-of(id I)

evaluate-mode [ E1 DO E2 ] =
      both  evaluate-mode E1 then
            coerce a val-mode from the result-mode then
            obtain a 1st operand-type from type-of(the val-mode)
      and   evaluate-mode E2 then
            coerce a val-mode from the result-mode then
            obtain a 2nd operand-type from type-of(the val-mode)
      then  typify-result DO then
            obtain a result-mode from
                  val-mode(the operator-result-type)

...

typify-result: Dyadic-Operator -> Action
!     'typify-result DO' checks the application of DO to
!     two operands.
!     It receives a 1st operand-type and a 2nd operand-type,
!     and it produces an operator-result-type.
!     It produces no bindings.

...
```

```
constrain: Statement -> Action
!     'constrain S' checks S.
!     It produces no values.
!     It produces no bindings.

constrain [ S1 ";" S2 ] =
      both  constrain S1
      and   constrain S2

constrain [ "while" E "do" S ] =
      both  evaluate-mode E then
            coerce a val-mode from the result-mode then
            check type-of(the val-mode) is boolean-type
      and   constrain S

constrain [ VD ";" "begin" S "end" ] =
      hide-locals VD before
      declare VD before
      constrain S

...

hide-locals: Variable-Declarations -> Action
!     'hide-locals VD' hides any non-local bindings of
!     identifiers declared in VD.
!     It produces no associations.
!     It produces bindings to 'undefined'.

...
```

```
declare: Variable-Declarations -> Action
!     'declare VD' checks the declarations in VD.
!     It produces no values.
!     It produces bindings.

declare [ "var" I ":" TY ] =
      first typify TY
      then  both   check binding-of(id I) is undefined
            and    bind id I to entire-var-mode(the type)

...
```

References

[1] Formal Definition of the Ada Programming Language, Preliminary Version. 1980.

[2] The Pascal Standard, ISO 7185. 1982.

[3] Reference Manual for the Ada Programming Language, ANSI/MIL-STD 1815 A. 1983.

[4] K. R. Apt. Ten years of Hoare's logic: A survey. In Proc. 5th Scand. Logic Symp., Aalborg Univ. Press, 1979.

[5] E. A. Ashcroft, M. Clint, and C. A. R. Hoare. Remarks on 'Program proving: Jumps and functions, by M. Clint and C. A. R. Hoare. Acta Inf., 6:317-318, 1976.

[6] E. Astesiano et al. On parameterized algebraic specification of concurrent systems. In Proc. CAAP-TAPSOFT 85, Springer-Verlag, 1985. LNCS 185.

[7] D. Bjørner and C. B. Jones. Formal Specification and Software Development. Prentice-Hall, 1982.

[8] D. Clement, J. Despeyroux, et al. Natural Semantics on the Computer. Rapport de Recherche No. 416, INRIA, 1985.

[9] E. W. Dijkstra. Guarded commands, non-determinacy, and formal derivations of programs. Commun. ACM, 18:453-457, 1975.

[10] P. Folkjær and D. Bjørner. A formal model of a generalized CSP-like language. In Proc. IFIP'80, North-Holland, 1980.

[11] K. Futatsugi, J. A. Goguen, et al. Principles of OBJ2. In Proc. POPL'84, ACM, 1984.

[12] M. J. C. Gordon. The Denotational Description of Programming Languages. Springer-Verlag, 1979.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12:576-580, 1969.

[14] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. Acta Inf., 2:335-355, 1973.

[15] R. Milner. The standard ML core language. Polymorphism, II(2), 1985.

[16] P. D. Mosses. Abstract semantic algebras! In Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts II (Garmisch-Partenkirchen, 1982), North-Holland, 1983.

[17] P. D. Mosses. A basic abstract semantic algebra. In Proc. Int. Symp. on Semantics of Data Types (Sophia-Antipolis), Springer-Verlag, 1984. LNCS 173.

[18] P. D. Mosses and D. A. Watt. Pascal: Action semantics. March 1986. Draft, Version 0.28.

[19] G. D. Plotkin. An operational semantics for CSP. In Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts II (Garmisch-Partenkirchen, 1982), North-Holland, 1983.

[20] G. D. Plotkin. A Structural Approach to Operational Semantics. DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[21] D. S. Scott and C. Strachey. Towards a Mathematical Semantics for Computer Languages. Tech. Mono. PRG-6, Programming Research Group, Oxford University, 1971.

[22] J. E. Stoy. The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977.

[23] R. D. Tennent. The denotational semantics of programming languages. Commun. ACM, 19:437-453, 1976.

[24] D. A. Watt. Executable semantic descriptions. Software: Practice and Experience, 16:13-43, 1986.

[25] P. Wegner. The Vienna definition language. ACM Comput. Surv., 4:5-63, 1972.