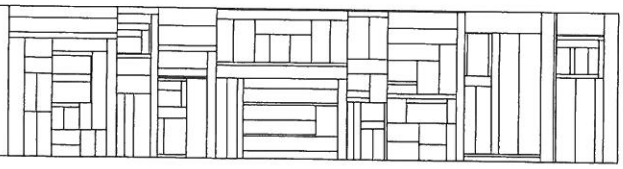
ISSN 0105-8517

The Use of Action Semantics

Peter D. Mosses
David A. Watt

DAIMI PB – 217
August 1986

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



THE USE OF ACTION SEMANTICS[1]

Peter D. Mosses

Computer Science Department
Aarhus University
DK-8000 Aarhus C, Denmark

David A. Watt

Computing Science Department
University of Glasgow
Glasgow G12 8QQ, U.K.

Formal descriptions of semantics have so far failed to match the acceptance and popularity of formal descriptions of syntax. Thus, in current standards for programming languages, syntax is usually described formally but semantics informally, despite the greater danger of impreciseness in the description of semantics. Possible reasons for this state of affairs are discussed. Action Semantics, which has been developed from Denotational Semantics and Abstract Semantic Algebras, has some features that may make it more attractive than other semantic formalisms. This paper describes and motivates Action Semantics, and gives some realistic examples of its use.

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 previous 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 (*e.g.*, [1,2]). 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 for describing conventional programming languages.

To appear in the *Proceedings of the IFIP TC2 Working Conference on Formal Description of Programming Concepts III* (Gl. Avernæs, Denmark, August 1986), published by North-Holland Publishing Company, Amsterdam.

Denotational Semantics [22,25,23,11] is perhaps the main contender. 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. This may be partly because the higher-order functions taken as denotations of phrases are unfamiliar objects to many. Moreover, the λ-notation used to express denotations has obscure operational implications, and it takes considerable time and effort to learn the various idioms of λ-programming (such as continuations) that are generally used in denotational descriptions. Lastly, little attention has been paid to modularization, so that, even in the absence of the other factors, larger descriptions would still be disproportionately difficult to grasp.

The Vienna Development Method (VDM) provides an alternative notation for denotational descriptions, usually referred to as "Meta-IV" [6]. In fact Meta-IV is an extension of λ-notation, incorporating a variety of familiar data types, such as trees, sets and mappings, and it even has imperative features. The operational implications of Meta-IV are more apparent than those of pure λ-notation, and there is less need to depend on idioms. However, the foundations of Meta-IV seem somewhat shaky, and there are the usual problems with the modularity and comprehensibility of large-scale descriptions.

The (largely) functional programming language ML [15] has also been tried as a meta-language for expressing denotational descriptions. The ML implementation enables the empirical testing of semantic descriptions (and rapid prototyping [26]). It is unclear how difficult it would be to reason about denotations expressed in ML.

It is also possible to use a general-purpose programming language as a meta-language: 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 (in some languages) of achieving modularity. However, such general-purpose programming languages, even when reduced to applicative subsets and extended to allow higher-order functions (which makes them less familiar!), turn out to be rather clumsy for expressing semantic descriptions. 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 [12,3] is reasonably accessible to programmers and implementors, involving only assertions about the values of variables, and the basic notions of formal logic (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 [13] does not cover the full language, and its rules for function declarations give logical inconsistency [4].

By the way, the so-called "weakest precondition" semantics of [8] is not particularly axiomatic, even though it uses assertions: it is denotational in essence, and suffers from some of the pragmatic deficiencies of the usual denotational approach.

Operational Semantics [27] 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 of operational descriptions.

The Structural Operational Semantics approach [21,20,7] employs axiomatic descriptions 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 [5], 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 significant 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 the 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 pragmatic 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 see how well it can cope with conventional programming languages. This experiment is nearing completion, and the results so far look promising [18].

The approach is called *Action Semantics*. It is based on "Abstract Semantic Algebras" [16,17], but the presentation of semantic descriptions has been significantly modified, with the aim of improving readability.

Action Semantics is *compositional*, like conventional Denotational Semantics (and Initial Algebra Semantics [10]), in that the semantics of each phrase is determined by the semantics of its subphrases. The difference is that the semantics of phrases (*i.e.*, their denotations) are no longer taken to be higher-order functions: they are "actions", which have (reasonably) simple operational interpretations, and quite nice algebraic properties.

In fact, Action Semantics may be regarded as "denotational", although taking actions as denotations generally gives a less "abstract" semantics than that obtained using higher-order functions — *i.e.*, fewer phrases are semantically-equivalent. (But note that "full abstractness" [19,14] is seldom achieved in conventional denotational descriptions, in practice.)

We consider actions in detail in the next section. 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. (Note that modularity alone does *not* necessarily give good modifiablity.)

Our 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 — we hope that this will encourage the casual reader to become a serious reader! There seem to be substantial pragmatic advantages, both for the readers and the authors of semantic descriptions, in having a notation with a fair amount of redundancy. Our action notation mimics natural language, but remains completely formal. (Other, more

"mathematical", representations of the notation could be adopted when conciseness is a primary concern.)

The formal interpretation of standard actions is specified by algebraic axioms. These 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 are to be used to "fine-tune" a previously-established conceptual understanding (and for formal reasoning about actions, of course).

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. But note that giving an algebraic specification of actions does not preclude supplementary specifications using other formalisms. For example, one could give a model for actions based on domains of higher-order functions — thus turning action-semantic descriptions into conventional denotational descriptions!

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 are then specified as mappings from context-free abstract syntax to actions, using the same notation.

In the rest of the paper, we first indicate the meta-notation used in Action Semantics, and explain the concept of actions. Then we give some examples of the use of Action Semantics, taken from the current version of our Pascal semantic description. We conclude by sketching our intentions regarding the future development of Action Semantics.

2. NOTATION

The overall organization of action-semantic descriptions is indicated in Table 1. The remaining tables and the appendices illustrate the various parts of an action-semantic description with examples taken from our Pascal description [18]. In this section we explain the notation used, leaving discussion of the Pascal examples themselves to the following sections.

By the way, informal comments may be used throughout action-semantic descriptions. Each comment is initiated by an exclamation mark '!', and terminated by the end of the line. (In this paper, we give explanatory comments in the text, rather than incorporating them in the examples.)

The first section of an action semantics presents the *standard notation* for values and actions. The sorts (*i.e.*, types) and operations of the notation are introduced using something akin to the OBJ2 algebraic specification framework [9]. We differ from OBJ2 mainly in blurring the distinction between sorts and "objects".

The standard *sorts* of values and actions are listed first, see Table 2. As in OBJ2, sorts may be *subsorts* of other sorts. They may also be parameterized by sorts (or values). A parameterized sort is identified with the union (*i.e.*, least supersort) of all its instances.

After the sorts come the *operations* on values and actions. The sorts of the arguments and result of each operation are given in the usual way, see Tables 3 and 4. "Mixfix" notation for operations is specified, as in OBJ2, using underlines '_' to indicate argument positions (prefix notation is assumed otherwise). Optional bits of notation are enclosed in square brackets.

Preface

1. Standard Notation

   1.1. Sorts
   1.2. Operations

2. Abstract Syntax

   2.1. Abstract Phrase Sorts
   2.2. Abstract Phrase Productions

3. Static Semantics

   3.1. Special Sorts
   3.2. Special Operations
   3.3. Semantic Functions

4. Dynamic Semantics

   4.1. Special Sorts
   4.2. Special Operations
   4.3. Semantic Functions

Appendices

A. Algebraic Axioms for Standard Notation
X. Explanation of Meta-Notation

TABLE 1
Organization of an Action Semantics

The *axioms* that formally specify the interpretation of the operations are deferred to an appendix (not illustrated in this paper). The form of the axioms is much as in OBJ2, allowing equations and positive conditional equations in terms over the operations and (sorted) variables.

The second section of an action semantics specifies the (context-free) *abstract syntax* of the programming language to be described. The style of presentation of abstract syntax is similar to that commonly used in conventional Denotational Semantics, *i.e.*, "abstract BNF". See Table 5 for an example. The *sorts* of abstract syntactic phrases are listed first, together with their relationship to the nonterminal symbols of the programming language's concrete grammar. (At present the relationship is stated only informally, by means of comments.) Then *productions* list the alternative constructions for each phrase sort, introducing sorted syntactic variables that are used as abstract nonterminals (these are later used also in the semantic equations). Note that the "abstract terminal symbols" used in abstract productions have no formal connection with concrete syntax: they are merely a suggestive means of distinguishing abstract constructs. Algebraically, each alternative of a production corresponds to the introduction of an operation with (possibly) mixfix syntax. Ambiguity in the abstract grammar is of no concern, as the grammar is not used for parsing strings of symbols, only for describing the (tree) structure of

abstract phrases.

The remaining section (or sections) specifies the semantic functions for the programming language. As a prelude, *special sorts and operations* may be introduced (see Tables 6 and 7). As well as new sorts being introduced here, sort parameters of the standard notation are instantiated to unions of other sorts, effectively specializing the standard notation for the purpose of the semantics of the particular language to be described. (This is analogous to identifying the "characteristic domains" in conventional Denotational Semantics: the domains of denotable, storable and expressible values [24].)

The *semantic functions*, mapping abstract phrases to actions (or, occasionally, to values) are specified inductively by semantic equations. See the appendices for examples. The notation is just as in conventional Denotational Semantics.

The left hand side of each semantic equation is the application of a semantic function to a schematic abstract phrase construction. The phrase construction is indicated by abstract nonterminals and terminal symbols, enclosed in double square brackets. Primes and/or subscripts may be used on the nonterminals so as to identify different subphrases of the same sort.

The right hand side is any term built from the (standard and special) operations and from applications of semantic functions to subphrases of the phrase construction given in the left hand side. In practice, we may relax this strict "homomorphic" discipline and allow compound constructions on the right hand side, provided that this does not affect the well-foundedness of the inductive definition. This can be used to emphasize that one phrase is merely an abbreviation for another phrase in the language.

Defining several semantic functions on a single sort abbreviates defining a single semantic function mapping to tuples (of values or actions). This is especially convenient for factorization into *static semantics* and *dynamic semantics*, as there is often not much commonality between them. The total semantics of programs can then be defined as a particular combination of their static and dynamic semantics.

We now explain and motivate our *standard notation* for actions and values. The formal syntax of the notation is given in Tables 2–4.

An *action*, conceptually, is just an entity that can be performed so as to process information. In general, the outcome of a performance may be *completion* (normal termination), or *escape* (exceptional termination), or *divergence* (non-termination), or *failure*.

When performed, actions operate on named values, cells of storage, and bindings for tokens. It is useful to consider separately various *facets* of actions: in the *functional* facet, actions receive and produce sets of named values; in the *imperative* facet, actions receive and may change the contents of cells in storage (they may also create and destroy cells); in the *binding* facet, actions receive and produce sets of bindings of tokens to values.

The various facets are *orthogonal*, in that operations in one facet do not interfere with operations in the other facets. For example, storing a value in a cell of storage does not produce new bindings for tokens; binding a token to a value does not cause any change to storage.

This separation of facets is closely related to the distinct uses in conventional Denotational Semantics of arguments, states, and environments. In the action notation, however, we do not refer explicitly to such data structures, in the interests of modifiability.

Boolean	Natural	List(S)	Set(S)
Value	Name(S)	Adjective	Term(S)
Storable	Cell(S)	Bindable	Token(S)
Undefined	S ?	Abstraction	Action

TABLE 2
Standard Sorts

Each facet has rather different characteristics with respect to the propagation of information. In the functional facet, values are *transient*, and are "forgotten", unless explicitly copied. In the imperative facet, in contrast, the values stored in cells are *stable*, remaining constant until new values are stored there (or the cells are destroyed). In the binding facet, the bindings are established for a particular *scope*, and earlier bindings get re-established when performance leaves a scope.

The consideration of facets of actions has guided the design of the standard action notation. Each primitive action is generally concerned with one particular facet, and does not produce information in the other facets. Likewise, each action combinator is essentially a tuple of basic single-faceted combinators, and usually one or more of these are neutral, not restricting the flow of information either into or out of sub-actions.

There are various standard *sorts* in the action notation. These are listed in Table 2. First of all, there are the usual sorts of mathematical values, such as Boolean and Natural. For any sort of values S, the sort List(S) is standard; also Set(S), when S has an equality operation.

For each sort of values S, the sort S? is the union of S with the sort Undefined, whose only value is "undefined". (Note that "undefined" is essentially an "error value" — it does *not* represent non-termination!)

Then there are the sorts Name, Cell, and Token. Names are used for referring to previously-computed values in actions (like bound variables in λ-notation). Cells are the identities (addresses, locations) of *potential* bits of storage; we assume that storage management is implementation-dependent, so the action that creates a cell may choose any cell that is not part of the current storage. Tokens are syntactic items such as identifiers and labels, to which values may be bound.

Associated with Name, Cell, and Token are the sorts Value, Storable, and Bindable, respectively. These sorts are not further specified in the standard notation, they are essentially parameters that can be instantiated with whatever particular (unions of) sorts are required for the semantics of a particular programming language.

It is convenient to assume subsorts Name(S), Cell(S), and Token(S) corresponding to the subsorts (S) of Value, Storable, and Bindable, respectively. Note that the subsorts Name(S) do not inherit the subsort relation of Value: in fact they are *disjoint*. Nevertheless, a name in Name(S) may still refer to a value that is in a subsort of S, as such a value is also of sort S.

The sort Term provides the means for actions to use operations on values. Terms may contain references to named values, the contents of cells, and the bindings of tokens — note that these references are *not* regarded as separate actions, in contrast to earlier versions of the action notation [16]. For each value sort S, Term has a subsort Term(S), consisting of terms that yield values of sort S (or the value "undefined").

```
undefined       :                          → Undefined

true            :                          → Boolean
false           :                          → Boolean
complement      : Boolean                  → Boolean
disjunction     : Boolean, Boolean         → Boolean
conjunction     : Boolean, Boolean         → Boolean

0               :                          → Natural
_ + 1           : Natural                  → Natural
 ...

empty-list      :                          → List(S)
only            : S                        → List(S)
join            : List(S), List(S)         → List(S)
first-of        : List(S)                  → S?
rest-of         : List(S)                  → List(S)?
length          : List(S)                  → Natural

empty-set       :                          → Set(S)
singleton       : S                        → Set(S)
union           : Set(S), Set(S)           → Set(S)
difference      : Set(S), Set(S)           → Set(S)
intersection    : Set(S), Set(S)           → Set(S)
is-equal        : Set(S), Set(S)           → Boolean
is-subset       : Set(S), Set(S)           → Boolean
is-contained    : S, Set(S)                → Boolean

s               :                          → Name(S)
s1-s2           :                          → Name(S2(S1))
_ _             : Adjective, Name(S)       → Name(S)

the _           : Name(S)                  → Term(S)
contents-of     : Term(Cell(S))            → Term(S)
binding-of      : Term(Token(S))           → Term(S)
op              : Term(S1), ..., Term(Sn)  → Term(S)

abstraction _   : Action                   → Term(Abstraction)
_ with the _    : Term(Abstraction), Name  → Term(Abstraction)
_ importing all : Term(Abstraction)        → Term(Abstraction)
```

TABLE 3

Standard Operations on Values and Terms

Finally, there are the standard sorts Action and Abstraction. Actions are not themselves values, but they can be encapsulated in *abstractions* (which *are* values) and subsequently enacted.

Now for the standard *operations*. The operations on values are listed in Table 3, and the suggestive words used in the operation symbols are intended to make further informal explanation superfluous.

The standard notation for names in Name(S) is to use the lower-case spelling "s" of the sort S, *e.g.*, "boolean" of sort Name(Boolean). Different names of the same sort are formed by prefixing adjectives, such as "1st", "2nd", "other" — adjectives should not clash with other symbols in the notation, but are otherwise arbitrary. When S is an instantiation of a parameterized sort, *e.g.*, List(Boolean), we allow the spelling of the parameter to prefix the spelling of the parameterized sort, *e.g.*, "boolean-list" (instead of "list(boolean)").

References to the values of names in terms are made by use of the definite article, "the". Thus "the boolean" in a term yields the value named by "boolean", if there is one; otherwise it yields the value "undefined". Analogously, "contents-of" yields the value stored in a cell, *e.g.*, "contents-of(the boolean-cell)"; and "binding-of" yields the value bound to a token, *e.g.*, "binding-of(*I*)", where *I* is an (identifier) token.

Value operations used in terms are strict in "undefined", so that the evaluation of a term containing references to undefined names, *etc.*, always yields "undefined".

We defer the explanation of the operations that involve abstractions until the end of the section.

Now for the standard primitive actions and action combinators. There is insufficient space here for full details; we consider only the main features of each form of action, ignoring the special cases that arise when actions are used in "unintended" ways. (To prohibit the unintended uses of actions syntactically would involve the use of a cumbersome notation for subsorts of actions, which we currently prefer to avoid.) The syntax of the standard action notation is given in Table 4.

The number of primitives and combinators in the standard notation may seem rather large — in fact there are 25 of them in all. But the conceptual complexity of the action notation is much less than this might suggest, thanks to the orthogonality (and commonality) of the various facets. Indeed we claim that all the actions express familiar computational notions. In any case, the aim is for the same action notation to be used in the semantic descriptions of many different programming languages, which may compensate for the effort of learning it.

Below, N stands for an arbitrary name, T for a term, and A for an action (all with optional subscripts). Note that the square brackets used are not symbols themselves, they merely enclose optional symbols.

Except where stated otherwise:

- actions complete (*i.e.*, terminate normally);
- the values, storage, and bindings received by actions are propagated to their component actions and terms;
- escapes, divergence and failures propagate;
- primitive actions produce no values, make no changes to storage, and produce no bindings.

"obtain a[n] N from T" names the value of T by N, if that value is of the sort indicated by N; otherwise it fails. *Usage:* Naming values for later reference, renaming values to avoid name clashes, and checking that values belong to particular subsorts.

Example: obtain a boolean from contents-of(the cell) .

"check T_1 is T_2" completes if the value of T_1 is the same as the value of T_2; otherwise it fails. *Usage:* Checking for a particular value, and guarding alternative actions.

Example: check the boolean is true .

obtain a[n] _ from _	: Name(S1), Term(S2)	→ Action
check _ is _	: Term(S1), Term(S2)	→ Action
copy the _	: Name(S)	→ Action
copy all	:	→ Action
skip	:	→ Action
fail	:	→ Action
create a[n] _	: Name(Cell(S))	→ Action
destroy _	: Term(Cell(S))	→ Action
store _ in _	: Term(S), Term(Cell(S))	→ Action
bind _ to _	: Term(Token(S)), Term(S)	→ Action
[either] _ or _	: Action, Action	→ Action
[preferably] _ else _	: Action, Action	→ Action
[both] _ and _	: Action, Action	→ Action
[first] _ then _	: Action, Action	→ Action
_ before _	: Action, Action	→ Action
_ within _	: Action, Action	→ Action
block _	: Action	→ Action
enact _	: Term(Abstraction)	→ Action
_ where □ is _	: Action, Action	→ Action
□	:	→ Action
recursively _	: Action	→ Action
escape	:	→ Action
_ then exceptionally _	: Action, Action	→ Action
_ and exceptionally _	: Action, Action	→ Action
error	:	→ Action

TABLE 4
Standard Operations on Actions

"copy the N" copies only the value named N. *Usage:* Propagating values.

Example: copy the 1st boolean .

"copy all" copies all the named values.

"skip" simply completes, forgetting all the named values.

"fail" simply fails.

"create a[n] N" creates a cell whose contents are initially "undefined". The sort of the contents is indicated by N. *Usage:* Allocating storage for (simple) variables.

Example: create a boolean-cell .

"destroy T" destroys the cell given by the value of T. *Usage:* Re-allocating storage.

Example: destroy the boolean-cell .

"store T_1 in T_2" stores the value of T_1 in the cell given by the value of T_2. *Usage:* Assigning to (simple) variables.

Example: store true in the boolean-cell .

"bind T_1 to T_2" binds the token given by the value of T_1 to the value of T_2. *Usage:* Declaring identifiers and formal parameters.

Example: bind id *I* to the boolean-cell .

"[either] A_1 or A_2" may choose between performing A_1 or performing A_2; but if the chosen action fails, it performs the other action instead. *Usage:* Combining alternative actions with exclusive guards, and implementation-dependent (non-deterministic) choice.

Example: either
check the boolean is true then *A'*
or check the boolean is false then *A"* .

"[preferably] A_1 else A_2" performs A_1; but if A_1 fails, it performs A_2 instead. *Usage:* Combining alternative actions with non-exclusive guards.

Example: preferably
obtain a boolean from the value then *A'*
else error .

"[both] A_1 and A_2" performs A_1 and A_2 together, combining any values and bindings that they produce, and interleaving any changes to storage that they make. *Usage:* Implementation-dependent combination of interfering actions, and ordinary combination of independent actions.

Example: both obtain a boolean from first-of the boolean-list
and obtain a boolean-list from rest-of the boolean-list .

"[first] A_1 then A_2" performs A_1, followed by A_2 (if A_1 completes). The only values produced by the whole action are those produced by A_2, and the only values received by A_2 are those produced by A_1. *Usage:* Sequential composition of actions.

Example: first create a boolean-cell
then bind id *I* to the boolean-cell .

"A_1 before A_2" performs A_1, followed by A_2 (if A_1 completes). In contrast to "A_1 then A_2", both A_1 and A_2 receive the values received by the whole action, and the whole action produces all the values produced by A_1 and A_2 (as in "A_1 and A_2"). The bindings produced by the whole action are those produced by A_1, overridden by those produced by A_2, which itself receives the bindings received by the whole action overridden by those produced by A_1. *Usage:* Sequential composition of declarations, and sequential evaluation of expressions.

Example: bind id I_1 to the 1st value before
bind id I_2 to the 2nd value .

"A_1 within A_2" performs A1, followed by A_2 (if A_1 completes). This is like "A_1 before A_2", except that bindings are composed differently: the only bindings produced by the whole action are those produced by A_2, and the only bindings received by A_2 are those produced by A_1. *Usage:* Restricting the scopes of declarations.

Example: bind id I_1 to the 1st value within
bind id I_2 to the 2nd value .

"block *A*" performs *A*, but then discards any bindings produced by *A*. *Usage:* Localizing the scopes of declarations in block structure.

Example: block *A'*
before *A"* .

"A_1 where □ is A_2" performs A_1. Whenever the performance reaches an occurrence of the dummy action "□", A_2 is performed in place of "□". This corresponds to performing the action given by replacing the occurrences of "□" in A_1 with the (perhaps infinite) action obtained by unfolding A_2 at "□". Note that "where" can be nested; but there does not seem to be a need for more than the one dummy action, "□". *Usage:* Iterating actions.

Example:
```
first obtain a natural from 42
then  □
where □ is
      preferably
            check the natural is 0 and copy the natural
      else  first obtain a natural from the natural - 1
            then  □
            then  obtain a natural from the natural + 1 .
```

"recursively A" performs A, which must complete, producing only bindings: no values, no changes to storage. The action A receives the bindings received by the whole action, overridden by those produced by A itself. *Usage:* Making declarations recursive.

Example:
```
block recursively A'
      before A" .
```

"escape" simply escapes, producing all the received values. *Usage:* Raising exceptions, and performing jumps.

Example:
```
copy the value then escape .
```

"A_1 then exceptionally A_2" performs A_1 first. If A_1 escapes, then A_2 is performed as well, the only values received being those produced by A_1, and the escape is not propagated. *Usage:* Handling exceptions and jumps to labelled statements.

Example:
```
A' then exceptionally
      preferably check the value is 0 else escape .
```

"A_1 and exceptionally A_2" performs A_1 first. If A_1 escapes, then A_2 is performed as well, the only values received being those received by the whole action, and the escape is propagated, producing all the values produced by A_1 and A_2. *Usage:* Adding further values to escapes.

Example:
```
A' and exceptionally
      first copy the extra value
      then  escape .
```

"error" escapes, not producing any values. *Usage:* Dynamic error conditions causing premature termination of programs.

Now we consider abstractions. The term "abstraction A" yields a value of sort Abstraction, encapsulating the action A. The action "enact T" performs the action encapsulated in the abstraction value given by evaluating T. The encapsulated action does *not* receive the values and bindings received by "enact T", although it does receive the storage.

Such simple abstractions could be used for parameterless procedures *without* non-local references to bindings (there being then no distinction between static and dynamic scope rules).

In order to deal with parameter-passing, we use the term "T with the N", where T yields an abstraction. The value of this term is an abstraction that encapsulates the same action that the value of T encapsulates, except that the action now receives the value named by N — the

same value as received by "T `with the` N" — when enacted. Thus the value named by N is "frozen" into the abstraction given by "T `with the` N".

Further named values can be supplied to the abstraction given by "T `with the` N" in the same way, but of course the value of N cannot be altered. This is adequate for dealing with the concept of partial application used in various programming languages. Note that normal (total) application is expressed by "`enact` T `with the` N", where N names a previously-computed list of actual parameter values.

Similarly, to specify whether static or dynamic scopes are to be used for non-local references to bindings in abstractions, we use the term "T `importing all`". This gives an abstraction in which all the bindings received by this term are also received by the encapsulated action. So "`(abstraction` A `) importing all`" gives an abstraction with static scopes, whereas "`enact` T `importing all`" gives dynamic scopes for the abstraction yielded by T, in general.

There does not seem to be much use for abstractions like "T `with all`" and "T_1 `importing` T_2" (where the value of T_2 is a token), but they could easily be incorporated in the full standard notation.

A final point about the notation, concerning disambiguation: It would be intolerable to insist on full parenthesization of actions, as the plethora of parentheses would greatly hinder the fluent reading of the notation. On the other hand, it would be just as unreasonable to expect the reader to remember an elaborate operator precedence.

Our solution is to use *indentation* as a suggestive way of indicating grouping. The main rule is that each (maximal) indented sequence of lines is always a complete action — this can be formalized by means of an attribute grammar, with an inherited attribute corresponding to indentation level. Within a sequence of lines at the same level of indentation, combinators associate to the left. By the way, *all* the combinators in the action notation are associative, so repeated combinations involving the same combinator do not need grouping at all.

When the nesting gets too deep for indentation, as it does occasionally in our Pascal semantics, we resort to punctuation: an operator preceded by a comma has lower precedence than any operator without punctuation marks, and higher precedence than operators preceded by semicolons (as in ordinary English sentence structure). We reserve parentheses for use in terms occurring in actions.

3. DYNAMIC SEMANTICS

In this section we discuss some examples of dynamic semantics. These examples are all taken from the current version of our action-semantic description of ISO Standard Pascal (Level 0) [18].

Table 5 gives our abstract syntax for some typical expressions, statements, and declarations in Pascal. The relation between this abstract syntax and the standard concrete syntax of Pascal is quite straightforward: the abstract syntax essentially just ignores those details of concrete syntax that have no semantic significance, *e.g.*, whether an expression E is a term or a factor.

Before the action notation can be used to specify the denotations of the abstract phrases of Pascal, it needs to be specialized with respect to the sorts of values that actions process. Table 6 gives the special sorts appropriate for Pascal. (Actually, a few sorts are omitted in Table 6 — they will be discussed at the end of this section.)

```
                Identifier              !   identifier
                Unsigned-Integer        !   unsigned-integer
                Expression              !   expression | variable-access |
                                        !   actual-parameter
                Expression-List         !   expression { , expression}
                Dyadic-Operator         !   adding-operator | ...
                Statements              !   statement | statement-sequence
                Block                   !   block
                Procedure-and-Function-Declaration-Part
                                        !   procedure-and-function-declaration-part
                Formal-Parameter-Part
                                        !   [formal-parameter-list]
                Formal-Parameter-Sections
                                        !   formal-parameter-section {;
                                        !           formal-parameter-section}
                Identifier-List         !   identifier-list

  I :   Identifier              !   productions omitted
 UI :   Unsigned-Integer        !   productions omitted

  E :   Expression          ::=   I | UI | E DO E | ...

 DO :   Dyadic-Operator     ::=   + | ...

  S :   Statements          ::=   E := E | S ; S | if E then S else S
                                |   while E do S | I ( EL ) | ...

 EL :   Expression-List     ::=   E | E , EL

  B :   Block               ::=   ... PFDP begin S end

PFDP :  Procedure-and-Function-Declaration-Part
                            ::=   procedure I FPP ; B ; | PFDP PFDP | ...

 FPP :  Formal-Parameter-Part
                            ::=   ( FPS ) | EMPTY

 FPS :  Formal-Parameter-Sections
                            ::=   IL : I | var IL : I | FPS ; FPS | ...

  IL :  Identifier-List     ::=   I | IL , IL
```

TABLE 5
Some of the Pascal Abstract Syntax

Some of the sorts in Table 6 are parameterized by sorts (ranged over by S) or integers (ranged over by M, N).

The sorts Character, Integer, and Real are all "implementation-defined", and they are parameters of the whole semantic description. (The constraints on their associated operations are to be specified formally in an appendix.)

Enumerand(N) is essentially a copy of the integers 0,...,N-1. String(N) has the same values as Array(1,N,Character), where Array(M,N,S) is the sort of tuples indexed by the integers M, ..., N, with components of sort S. Record values are essentially mappings from (field-)identifiers to component values.

```
Character       Enumerand(N)      Integer
Real            String(N)         Array(N,N',S)        Record
Pointer(S)      File(S)           Variable(S)          Type(S)
Procedure       Function          Active-Function      Formal-Mode

Ordinal              = Boolean, Character, Enumerand, Integer
Storable             = Boolean, Character, Enumerand, Integer,
                       Real, Set, Pointer
Assignable           = Boolean, Character, Enumerand, Integer,
                       Real, Set, String, Array, Record, Pointer, File
Identifiable         = Boolean, Character, Enumerand, Integer,
                       Real, String, Variable, Type,
                       Procedure, Function, Active-Function
Token(Identifiable)  = Identifier
Bindable             = Identifiable, ...
Operand              = Boolean, Character, Enumerand, Integer,
                       Real, Set, String, Pointer
Operator-Result      = Boolean, Integer, Real, Set
Argument             = Boolean, Character, Enumerand, Integer,
                       Real, Set, String, Array, Record, Pointer,
                       Variable, Procedure, Function
Function-Result      = Boolean, Character, Enumerand, Integer,
                       Real, Pointer
Constant             = Boolean, Character, Enumerand, Integer,
                       Real, String
Result               = Boolean, Character, Enumerand, Integer,
                       Real, Set, String, Array, Record, Pointer,
                       Variable, Procedure, Function, Active-Function
Value                = ...
```

TABLE 6

Some Special Sorts for the Pascal Dynamic Semantics

Variable(S) is rather like the standard sort Cell(S), but it is generalized to deal with compound variables that may share cells in storage. Values of sort Type(S) merely give adequate structural information for allocating elements of Variable(S).

Procedure and Function values are really just pairs with components of sorts List(Formal-Mode) and Abstraction. The Formal-Mode values indicate whether coercions are to be applied to actual parameters. Each value of sort Active-Function consists of an ordinary Function value paired with a result-variable.

The remaining sorts are just unions of the above sorts. The sort Ordinal corresponds to the Pascal notion of ordinal values. Storable indicates what Cell values may contain, and Assignable does the same for Variable. Identifiable values can be bound to Pascal identifiers (the other sorts of Bindable values in the full Pascal semantics will be discussed later).

Operand values are operated upon by relational, Boolean and arithmetic operators, yielding values of sort Operator-Result. Similarly, Argument values are passed to procedures and functions as the values of actual parameters, and function activations (may) yield values of

sort Function-Result. Result encompasses all the values that can be yielded by expression[1] evaluation in Pascal; likewise Constant for constant values. Value is the union of all the (standard and special) value sorts, and indicates what can be referred to by names in the action notation.

A closer inspection of Table 6 reveals that no two of the various union sorts coincide! This indicates a certain semantic irregularity in Pascal, presumably due to implementation considerations and incidental syntactic restrictions — although it seems strange that Set values may be operands, operator-results, or arguments, but not function-results!

A few of the operations associated with the special sorts are listed in Table 7. Note particularly that some names (for instance, "procedure"), are overloaded and used as constructor operations.

The special operations "allocate", "assign", and "access" are generalizations of the Cell operations "create", "store", and "contents-of", respectively, to cope with compound variables. But note that "access" is an *action*, rather than a Term constructor: we do not assume that obtaining the value of a compound variable is an indivisible action. Formally, "allocate", "assign", and "access" are to be specified as abbreviations for compound standard actions involving (multiple) operations on cells.

Also, "coerce *N* from *T*" is an abbreviation for a compound action: it is just the same as "obtain *N* from *T*" when *T* yields a value of the same sort as the name *N*; otherwise, "coerce" performs Pascal-specific actions in order to get a value of the desired sort. Some of the possibilities are floating an integer to a real number, accessing the value of a variable, or calling a parameterless function. "coerce" is widely used in the Pascal action semantics.

We hope that the reader will now find it possible to get the gist of the semantic equations given in Appendix A — or even the full Pascal description. Even though the accuracy of the reader's understanding of the action notation may be poor to start with, it should increase rapidly by considering the actions corresponding to familiar Pascal constructs. Ultimately, points of doubt about the intended interpretation of the action notation are to be resolved by study of its algebraic axioms.

To help the reader, we provide the following comments on the semantic functions used in Appendix A.

For identifiers, "id *I*" is just *I* with all letters converted to lower case. Almost as trivial are the denotations of unsigned integer literals: "valuation *UI*" is the corresponding integer value. These denotations do not involve actions at all.

Expressions are more interesting. There are several reasons for letting the denotations of expressions be actions, rather than mere values. First, we wish to indicate explicitly that the order of evaluation of sub-expressions in Pascal is implementation-dependent; with actions, this is done by using the combinator "and".[2] (If evaluation of expressions were sequential, we would use "before" instead.) Next, function activations in expressions require the use of actions — and may have side-effects. Finally, our algebraic framework is first-order, so the denotations of operators occurring in expressions cannot be operations (on values), whereas they can be actions.

Anyway, "evaluate *E*" is an action that computes the *uncoerced* value of *E*, naming it

[1] In our abstract syntax, expressions include variable-accesses and actual-parameters.

[2] Actually, the Standard Pascal allows the evaluation of sub-expressions to be omitted (when their values are irrelevant) or to be done "in parallel"; but it is not clear to us exactly what that might mean.

```
zero                  :                                    → Integer
maximum-integer       :                                    → Integer
negation              : Integer                            → Integer
sum                   : Integer, Integer                   → Integer?
ordinal-number        : Ordinal                            → Integer
float                 : Integer                            → Real?
negation              : Real                               → Real
sum                   : Real, Real                         → Real?

procedure             : List(Formal-Mode), Abstraction     → Procedure
body-of               : Procedure                          → Abstraction
formal-mode-list-of   : Procedure                          → List(Formal-Mode)
function              : List(Formal-Mode), Abstraction     → Function
body-of               : Function                           → Abstraction
formal-mode-list-of   : Function                           → List(Formal-Mode)
active-function       : Function, Variable                 → Active-Function
function-of           : Active-Function                    → Function
result-variable-of    : Active-Function                    → Variable
value-mode            :                                    → Formal-Mode
variable-mode         :                                    → Formal-Mode
procedural-mode       :                                    → Formal-Mode
functional-mode       :                                    → Formal-Mode

allocate a[n] _ of _  : Name(Variable), Term(Type)         → Action
assign _ to _         : Term(Assignable), Term(Variable)   → Action
access a[n] _ in _    : Name(Assignable), Term(Variable)   → Action
coerce a[n] _ from _  : Name(S1), Term(S2)                 → Action
```

coerce a[n] *N* from *T* =

```
    preferably
        obtain a[n] N from T
    else obtain an integer from T then coerce a[n] N from the integer
    else obtain a variable from T then access a[n] N in the variable
    else ...
```

TABLE 7

Some Special Operations for the Pascal Dynamic Semantics

"result". For dyadic operators, "operate *DO*" is an action that receives two values named "1st operand" and "2nd operand", and produces a value named "operator-result".

The denotations of statements, given by "execute *S*", are actions that neither receive nor produce named values.

The evaluation of actual parameters by "evaluate-arguments *EL*" receives a list of formal mode values, and produces a list of argument values.

For blocks, "activate *B*" executes the declarations and statements of *B*, using "establish *PFDP*" to bind the declared procedure and function identifiers to appropriate values. (For brevity, we are ignoring other kinds of declarations occurring in blocks.) Finally, for formal

parameters, "formal-modes-of *FPP*" is the list of their modes (value, variable, *etc.*) — this is not an action, as it does not involve any processing of computed information — and "establish-formals *FPP*" binds the identifiers in *FPP* to the components of the argument-list received.

Finally, we consider how to deal with the action-semantic description of some of the "nastier" features of Pascal: forward procedure declarations, variant records, and jumps. Lack of space prohibits us from giving the formal description of these constructs here, the energetic reader is referred to [18].

Note, by the way, that we do *not* consider it to be a disadvantage of Action Semantics that it is able to cope with nasty, as well as nice, features of programming languages. The nastier a construct, the more it needs a formal description! However, we do hope that language designers might be discouraged from including nasty constructs by the complexity of their action-semantic descriptions. Approaches to formal semantics that lack generality and prohibit various features run the risk of alienating language designers, and could delay the development of new programming concepts.

Now for forward procedure declarations. These allow procedure headings to be declared separately from (but before) the corresponding procedure bodies. A heading declaration is an ordinary procedure declaration where the body is replaced by the symbol **forward**. In the body declaration, the procedure heading is abbreviated by leaving out the formal parameters. (Similarly for forward function declarations.)

The idea of forward declarations is to permit mutually recursive procedures while keeping to a "declaration-before-use" discipline. There is no problem in expressing mutual recursion in action semantics, the only difficulty is in reconstituting complete procedures from the separate headings and bodies.

Of course, one could just regard forward declarations as something to be eliminated in going from concrete syntax to abstract syntax. But that is not as trivial as it might sound, and cannot easily be separated from static semantics. Anyway, we prefer to treat the entire Pascal language in the dynamic semantics, rather than insisting that the reader be aware of various syntactic transformations and checks that are to be applied to programs before taking their semantics.

Briefly, our semantics for a forward-declaration of a procedure identifier *I* binds a token "forward-id *I*" to a value of sort Procedure-Head, consisting of a Formal-Mode-List value and an abstraction. The action encapsulated in the abstraction binds the formal-parameters of the procedure-heading to the arguments it receives, but there is no body yet to be the scope of these bindings. Subsequently, the body declaration for *I* binds "id *I*" to a normal Procedure value whose abstraction component is supplied with the abstraction component of the procedure head. The insertion of "recursively" in the semantic equation for block activation then gives the required mutual recursion between bindings for "id *I*".

An inspection of the semantic equations for forward declarations given in [18] indicates that this feature complicates the semantics of procedure declaration about threefold.

As for variant records, we hide most of the gory details in the special actions "allocate", "assign", and "access". The main idea is to represent fields of a variant record by abstractions which, when enacted, perform the required checks on the associated selector variable of the variant. Note that the Pascal with-statement rules out handling variant field accesses in the semantics of field-designator expressions, as it causes field identifiers to be bound directly to component variables. When changing variants, the old variant part gets destroyed, and the new variant part is created.

Finally, jumps and labels. In conventional Denotational Semantics, the idiom of "continuations" is used to express sequential performance. This allows labels to be bound to continuations that represent "the rest of the computation" from the label onwards. Denotations of statements, *etc.*, take continuations as arguments, and the goto-statement merely ignores its continuation argument and applies the continuation bound to the label. This is not possible in the standard action-notation, where sequencing is expressed directly, rather than by means of idioms.

Instead, we bind labels to dynamically-generated values of sort `Jump-Point`. The only feature of these values is that they can be tested for equality. The execution of a goto-statement just escapes with the jump-point bound to the label. The semantics of block activation is extended to include, after the execution of the statements of the block, the exceptional performance of an auxiliary semantics of statements. This handles any escapes with those jump-points that are bound to the labels in the statements. (This has to be iterated, using the "`where`" notation, to cater for subsequent jumps at the same block-level.) The idea of using escapes instead of continuations has been exploited in VDM [6]; we were led to it by consideration of the algebraic properties of actions.

An alternative approach to jumps and labels would be to let the jump-points include abstractions that encapsulate the performance of the rest of the statements in the block. Then a jump escapes to the activation level indicated by the jump-point, executes the (recursively-bound) abstraction and finally terminates the block. But this turns out to be even more complicated to specify than the approach sketched above.

For more examples of Pascal nasties, see [18]!

4. STATIC SEMANTICS

Action Semantics was designed originally to describe the *dynamic semantics* of (conventional) programming languages — witness the terminology 'action' and the selection of standard actions (Table 4). In developing our action-semantic description of Pascal, however, we have found that Action Semantics is quite suitable for describing *static semantics* too.

Because the standard actions are worded as imperative verbs and conjunctions, the dynamic semantic description reads like a set of instructions stating what has to be done when each program phrase is 'executed'. The static semantic description reads like a set of instructions stating what has to be done to check whether each program phrase is well-formed or not (in a context-sensitive sense).

Of course, there are important differences between the static and dynamic semantic descriptions of (conventional) programming languages. In the Pascal description we find the following differences:

- The values processed by the dynamic semantics are integers, reals, sets, pointers, variables, procedures, functions, and so on. The values processed by the static semantics are constants, types, 'variable-modes', 'procedure-modes', 'function-modes', and so on. Consider variables, for example: the dynamic semantics is concerned principally with the storage cell(s) occupied by each variable; whereas the static semantics is concerned principally with its type, and sometimes with whether it is a component of a packed structure. We use the term 'mode' for the statically-known properties of an entity such as a variable, procedure, function, or formal parameter.

Some values are common to the dynamic semantics and the static semantics, such as constants and types — although types are used for type-checking in the static semantics but only for storage allocation of variables in the dynamic semantics.

- Actions concerned with storage — namely "create", "destroy" and "store" — are not ordinarily used in the static semantics. To that extent, the entire imperative facet of actions is redundant in the static semantics. (One exception is mentioned later in this section.)
- Iteration and escapes are never used in the static semantics, so the actions "_ where □ is _", "escape", "_ then exceptionally _" and "_ and exceptionally _" are not needed.

A consequence of the last point is that no action in the Pascal static semantics ever diverges or escapes; the only possible outcomes are completion and failure. We adopt the following convention: if the denotation of a phrase is an action that completes, that phrase is well-formed; if the denotation of a phrase is an action that fails, that phrase is ill-formed.

In the static semantics of Pascal, the denotation of a program P is "constrain P", which completes if and only if P is well-formed. In the dynamic semantics, the denotation of P is "run P", which when performed will produce some changes in storage (*i.e.*, the storage where the external entities are held). These are composed to form the complete semantics of Pascal as follows, with the "then" combinator ensuring that "run P" is not performed if "constrain P" fails:

"constrain P then run P"

Now let us examine in more detail the static semantics of parts of Pascal whose abstract syntax is shown in Table 5. Some, but not all, of the standard sorts and operations (Tables 2-4) are used in the static semantics.

Some of the special sorts required by the Pascal static semantics are shown in Table 8, and some of the special operations in Table 9.

Sorts such as Integer–Type, Set–Type, and Pointer–Type correspond to the various type classes of Pascal. Some of these type classes contain a single type, *e.g.*, "integer–type" of sort Integer–Type. For each other type class an appropriate constructor operation is defined, *e.g.*, "set–type". Type is defined to be the union of all the type classes, and Ordinal–Type, Structured–Type, *etc.*, are defined to be unions of certain type classes only. Operations such as "is–numeric" and "assignment–compatible" correspond to the various predicates on types required in the static semantics.

Other sorts in Table 8 correspond to the various mode classes of Pascal. For example, Procedure–Mode is the sort of 'procedure-modes', each of which is constructed from a list of 'formal-modes' by the operation "procedure–mode". Our phrase sort Expression covers not only ordinary expressions (whose modes are of sort Data–Mode), but also variable-accesses (Variable–Mode), procedure identifiers (Procedure–Mode), and function identifiers (Function–Mode). The sort Result–Mode is defined to be the union of all these sorts. The operation "type–of" is overloaded on various mode classes such as Variable–Mode and Data–Mode.

Identifiable is instantiated for the Pascal static semantics to be the union of Constant, Type, Variable–Mode, Procedure–Mode, and Function–Mode. These are the 'values' to which identifiers are bound in the static semantics. Compare the instantiation of Identifiable in the dynamic semantics (Table 6).

Some semantic equations of the Pascal static semantics are shown in Appendix B.

Integer	Real	String	Packing
Integer-Type	Set-Type	Array-Type	Pointer-Type
Data-Mode	Procedure-Mode	Function-Mode	Active-Function-Mode

```
Constant                = Boolean, Character, Enumerand, Integer,
                          Real, String
Ordinal-Type            = Boolean-Type, ..., Integer-Type
Structured-Type         = Set-Type, Array-Type, ...
Operand-Type            = Boolean-Type, ..., Integer-Type, ..., Pointer-Type
Operation-Result-Type   = Boolean-Type, Integer-Type, ...
Function-Result-Type    = Ordinal-Type, Real-Type, Pointer-Type
Type                    = Ordinal-Type, Real-Type, Structured-Type, Pointer-Type
Formal-Mode             = Formal-Value-Mode, Formal-Variable-Mode, ...
Variable-Mode           = Entire-Variable-Mode, Component-Variable-Mode, ...
Result-Mode             = Constant, Data-Mode, Variable-Mode,
                          Procedure-Mode, Function-Mode
Identifiable            = Constant, Type, Variable-Mode,
                          Procedure-Mode, Function-Mode
Token(Identifiable)     = Identifier
Bindable                = Identifiable, ...
Value                   = ...
```

TABLE 8

Some Special Sorts for the Pascal Static Semantics

```
integer-type          :                                      → Integer-Type
real-type             :                                      → Real-Type
set-type              : Type-Name, Packing, Ordinal-Type     → Set-Type
is-numeric            : Type                                 → Boolean
assignment-compatible : Type, Type                           → Boolean
data-mode             : Type                                 → Data-Mode
procedure-mode        : List(Formal-Mode)                    → Procedure-Mode
type-of               : Data-Mode                            → Type
type-of               : Variable-Mode                        → Type
formal-modes-of       : Procedure-Mode                       → List(Formal-Mode)
formal-modes-of       : Function-Mode                        → List(Formal-Mode)
result-type-of        : Function-Mode                        → Function-Result-Type

coerce a[n] _ from _  : Name(Data-Mode), Term(Result-Mode)   → Action
```

TABLE 9

Some Special Operations for the Pascal Static Semantics

For expressions, "`infer-mode` *E*" is an action that infers the (uncoerced) mode of *E*, naming it "`result-mode`", if *E* is well-formed — otherwise it fails. The action "`typify-result` *DO*" checks the application of *DO* to two operands; it receives values named "`1st operand-type`" and "`2nd operand-type`", and produces a value named "`operation-result-type`" (unless it fails).

For statements, "`constrain` *S*" is an action that simply completes if *S* is well-formed — otherwise it fails. Its semantic equations illustrate the basic techniques of using Action Semantics to describe static semantics.

The denotations of blocks and declarations in Appendix B show how we enforce the language's scope rules. Declarations in Pascal are essentially sequential, so the combinator "`before`" is suitable in the way it accumulates bindings. However, no identifier *I* may be declared twice in the same block, nor may *I* be used in a block before its declaration (even if *I* is also declared in an enclosing block). Our solution deals with both constraints at once. The denotation (component) "`hide-locals` *B*" produces a binding of *I* to "`undefined`" for every identifier *I* declared in block *B*. Moreover, it fails if any identifier is declared more than once in *B*. Wherever "`declare-and-constrain` *B*" is performed, "`hide-locals` *B*" is performed beforehand. This prevents duplicate declarations of *I*. It also ensure that "`binding-of (id` *I*`)`" is "`undefined`" throughout the part of *B* that precedes the declaration of *I*.

For procedure and function declarations, "`declare` *PFDP*" binds all the identifiers declared in *PFDP* to their modes; "`produce-formal-mode-list` *FPP*" checks the formal-parameter-part *FPP* and deduces its formal-mode-list; and "`declare-formals` *FPP*" binds the formal parameters of *FPP*, in sequence, to the received formal-mode-list, for the purpose of checking whether the corresponding procedure body is well-formed.

The parts of the Pascal static semantics that we have illustrated are fairly straightforward. Some other aspects of Pascal, such as the restrictions on mutual recursion of types, forward procedures and functions, and scopes of labels, make the static semantic equations messy but are quite tractable.

One problem that is fairly troublesome is Pascal's name-equivalence rule for types. This requires a unique "type-name" to be generated for each anonymous type-denoter in the program. This is why (surprisingly) we do use the imperative facet in the Pascal static semantics. We can treat a type name as a kind of storage cell, whose content is an (unchanging) type. (Alternatively, we could let `Cell(Undefined)` be a subsort of `Token`, and bind type names to types.) Something of the sort would seem to be needed to formalize the name-equivalence rule in any denotational framework.

5. CONCLUSION

Action Semantics is a novel and (at least for the authors) exciting approach to the formal description of programming languages. Its novelty lies mainly in the standard notation for actions — hence the name. Other significant innovations are the use of formalized naturalistic language in the meta-notation, and the modular organization of action-semantic descriptions.

Action Semantics combines several techniques. The denotational technique is used to express the semantics of program phrases in terms of actions. The (algebraic) axiomatic technique is used to specify the intended interpretation of actions formally, but this is supplemented by an informal operational description of actions. We claim that this makes the most appropriate

use of the various techniques: for instance, it would not be appropriate to try to give algebraic axioms for unruly programming languages, nor would it give much insight into actions to reduce them to higher-order functions.

By the way, although we hope that the standard action notation will remain stable for some time to come, it may be useful to isolate special-purpose sub-notations, e.g., for use in static semantics. Note also that there is no guarantee that our action combinators can express all possible ways of combining actions: it may be necessary to add new ones, perhaps corresponding to concepts that we have overseen, or else just different ways of combinbing single-faceted combinators.

How do action-semantic descriptions compare to the usual *informal* semantic descriptions used in standard documents and reference manuals for programming languages? Well, despite the fact that informal semantic descriptions often use a rather legalistic and stilted language, we have to admit that they still *read* more fluently than our action notation. But on the other hand, when it comes to *understanding* the consequences of semantic descriptions, the informal reader of an action semantics should benefit from its compositional structure and uniformity of notation.

It will be interesting to see whether or not the programming community does find Action Semantics sufficiently useful to attract it away from informal descriptions. It seems to have become firmly established that formal semantics is generally incomprehensible to the average programmer. Action Semantics has tried to find a "middle way" between the rigours of formalism and the pitfalls of informal descriptions — but without abandoning formality! Instead of insisting that programmers should get to know an unfamiliar semantic universe of higher-order functions, we have tried to formalize *their* universe of actions, in the hope of a reconciliation between theory and practice.

We have worked out one large-scale example of the use of Action Semantics: a description of ISO Standard Pascal. We intend to provide other examples in the near future, such as Standard ML, in order to demonstrate that action-semantic descriptions are not particularly biased towards any one style of language — only towards good design and semantic regularity!

If there is sufficient interest in action-semantic descriptions for sequential languages, it will encourage us to extend actions with a communication facet, with a view to describing languages involving concurrency. Other action-semantics-based projects include compiler-generation and program development. But these may not be worthwhile unless Action Semantics becomes rather popular.

We appeal for reactions to Action Semantics: please let us know what could be improved, preferably with concrete examples. Also we would welcome collaboration in giving action-semantic descriptions of various programming languages. If there is enough interest, PDM will maintain a mailing list of "activists". By the way, our electronic addresses are: `pdm@daimi.uucp`, and `daw@cs.glasgow.ac.uk`.

ACKNOWLEDGMENTS

We thank our colleagues at home and abroad for encouragement during the early days of Action Semantics. The diverse comments from the referees on our extended abstract helped us to decide the contents of the full paper.

REFERENCES

[1] *The Pascal Standard, ISO 7185*. 1982. See [28].

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

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

[4] 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.

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

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

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

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

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

[10] 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.

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

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

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

[14] R. Milner. Fully abstract models of typed lambda-calculus. *Theoretical Comput. Sci.*, 1–22, 1977.

[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. August 1986. Draft, Version 0.3.

[19] G. D. Plotkin. LCF considered as a programming language. *Theoretical Comput. Sci.*, 5:223–255, 1977.

[20] 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.

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

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

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

[24] C. Strachey. *The Varieties of Programming Language.* Tech. Mono. PRG-10, Programming Research Group, Oxford University, 1973.

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

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

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

[28] I. R. Wilson and A. M. Addyman. *A Practical Introduction to Pascal — with BS 6192.* Macmillan, 1982.

APPENDIX A. SOME DYNAMIC SEMANTIC FUNCTIONS FOR PASCAL

```
id: Identifier → Identifier
```

...

```
valuation: Unsigned-Integer → Integer?
```

...

```
evaluate: Expression → Action
```

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

evaluate [[ UI ]] =
      obtain a result from valuation UI

evaluate [[ E1 DO E2 ]] =
      first 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 [[ + ]] =

      preferably
            obtain a 1st integer from the 1st operand and
            obtain a 2nd integer from the 2nd operand, then
            obtain an operator-result from
                  sum (the 1st integer, the 2nd integer)
            else  error
      else  coerce a 1st real from the 1st operand and
            coerce a 2nd real from the 2nd operand, then
            obtain an operator-result from
                  sum (the 1st real, the 2nd real)
            else  error
      or    obtain a 1st set from the 1st operand and
            obtain a 2nd set from the 2nd operand, then
            obtain an operator-result from
                  union (the 1st set, the 2nd set)
```

execute: Statements → Action

```
execute [[ E1 := E2 ]] =

      first both  evaluate E1 then
                  obtain a variable from the result
            and   evaluate E2 then
                  coerce an assignable from the result
      then  assign the assignable to the variable


execute [[ S1 ; S2 ]] =

      first execute S1 then execute S2


execute [[ if E then S1 else S2 ]] =

      evaluate E then
      coerce a boolean from the result then
      either
            check the boolean is true then
            execute S1
      or    check the boolean is false then
            execute S2
```

```
execute [[ while E do S ]] =

      □ where □ is
            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 [[ I ( EL ) ]] =

      first obtain a procedure from binding-of (id I)
      then  both  obtain an abstraction from body-of (the procedure)
            and   obtain a formal-mode-list from
                        formal-mode-list-of (the procedure) then
                  evaluate-arguments EL
      then  enact the abstraction with the argument-list
```

evaluate-arguments: Expression-List → Action

```
evaluate-arguments [[ E ]] =

      first evaluate E and
            obtain a formal-mode from first-of (the formal-mode-list)
      then  either
                  check the formal-mode is value-mode and
                  coerce an assignable from the result, then
                  obtain an argument from the assignable
            or    check the formal-mode is variable-mode and
                  obtain a variable from the result, then
                  obtain an argument from the variable
            or    ...
      then  obtain an argument-list from only (the argument)


evaluate-arguments [[ E , EL ]] =

      both  obtain a formal-mode-list from
                  only (first-of (the formal-mode-list)) then
            evaluate-arguments [[ E ]] then
            obtain an argument from first-of (the argument-list)
      and   obtain a formal-mode-list from
                  rest-of (the formal-mode-list) then
            evaluate-arguments EL
      then  obtain an argument-list from
                  join (only (the argument), the argument-list)
```

activate: Block → Action

activate [[... *PFDP* **begin** *S* **end**]] =

```
block ... before
            establish PFDP
      before
            execute S
```

establish: Procedure-and-Function-Declaration-Part → Action

establish [[**procedure** *I FPP* ; *B* ;]] =

```
recursively
      bind id I to procedure (
            formal-modes-of FPP,
            abstraction (
                  block establish-formals FPP before
                        activate B
                        ) importing all )
```

establish [[$PFDP_1$ $PFDP_2$]] =

```
establish PFDP1 before
establish PFDP2
```

formal-modes-of: Formal-Parameter-Part → List(Formal-Mode)

...

establish-formals: Formal-Parameter-Part → Action

establish-formals [[(*FPS*)]] =

```
establish-formals FPS
```

establish-formals [[*EMPTY*]] =

```
skip
```

establish-formals: Formal-Parameter-Section → Action

```
establish-formals [[ I : I' ]] =

      first both  obtain an assignable from first-of (the argument-list)
            and   obtain a type from binding-of (id I') then
                  allocate a variable of the type
      then  both  bind id I to the variable
            and   assign the assignable to the variable
      and   obtain an argument-list from rest-of (the argument-list)


establish-formals [[ IL1 , IL2 : I' ]] =

      establish-formals [[ IL1 : I' ; IL2 : I' ]]


establish-formals [[ var I : I' ]] =

      first obtain a variable from first-of (the argument-list)
      then  bind id I to the variable
      and   obtain an argument-list from rest-of (the argument-list)


establish-formals [[ var IL1 , IL2 : I' ]] =

      establish-formals [[ var IL1 : I' ; var IL2 : I' ]]


establish-formals [[ FPS1 ; FPS2 ]] =

      establish-formals FPS1 then
      establish-formals FPS2
```

APPENDIX B. SOME STATIC SEMANTIC FUNCTIONS FOR PASCAL

infer-mode: Expression → Action

```
infer-mode [[ I ]] =

      obtain a result-mode from binding-of (id I)

infer-mode [[ E1 DO E2 ]] =

      both  infer-mode E1 then
            coerce a data-mode from the result-mode then
            obtain a 1st operand-type from type-of (the data-mode)
      and   infer-mode E2 then
            coerce a data-mode from the result-mode then
            obtain a 2nd operand-type from type-of (the data-mode)
      then  typify-result DO
      then  obtain a result-mode from data-mode (the operation-result-type)
```

typify-result: Dyadic-Operator → Action

typify-result [[+]] =

preferably
check the 1st operand-type is integer-type and
check the 2nd operand-type is integer-type, then
obtain an operation-result-type from integer-type
else check is-numeric (the 1st operand-type) is true and
check is-numeric (the 2nd operand-type) is true, then
obtain an operation-result-type from real-type
or ...

constrain: Statements → Action

constrain [[E_1 := E_2]] =

both infer-mode E_1 then
obtain a variable-mode from the result-mode then
obtain a 1st type from type-of (the variable-mode)
and infer-mode E_2 then
coerce a data-mode from the result-mode then
obtain a 2nd type from type-of (the data-mode)
then check assignment-compatible (the 1st type, the 2nd type) is true

constrain [[S_1 ; S_2]] =

both constrain S_1
and constrain S_2

constrain [[**while** E **do** S]] =

both infer-mode E then
coerce a data-mode from the result-mode then
check type-of (the data-mode) is boolean-type
and constrain S

constrain [[I (EL)]] =

obtain a procedure-mode from binding-of (id I) then
obtain a formal-mode-list from formal-modes-of (the procedure-mode) then
modulate EL

modulate: Expression-List → Action

...

declare-and-constrain: Block → Action

declare-and-constrain [[... *PFDP* **begin** *S* **end**]] =

... before
declare *PFDP* before
constrain *S*

declare: Procedure-and-Function-Declaration-Part → Action

declare [[**procedure** *I FPP* ; *B* ;]] =

first produce-formal-mode-list *FPP*
then bind id *I* to procedure-mode (the formal-mode-list) before
block hide-locals *B* before
declare-formals *FPP* before
declare-and-constrain *B*

declare [[$PFDP_1$ $PFDP_2$]] =

declare $PFDP_1$ before
declare $PFDP_2$

hide-locals: Block → Action

hide-locals [[... *PFDP* **begin** *S* **end**]] =

... and hide-locals *PFDP*

hide-locals: Procedure-and-Function-Declaration-Part → Action

hide-locals [[**procedure** *I FPP* ; *B* ;]] =

bind id *I* to undefined

hide-locals [[$PFDP_1$ $PFDP_2$]] =

hide-locals $PFDP_1$ and hide-locals $PFDP_2$

produce-formal-mode-list: Formal-Parameter-Part → Action

...

declare-formals: Formal-Parameter-Part → Action

...