# An Introduction to Action Semantics

Peter D. Mosses

Computer Science Department, Aarhus University,

DK–8000 Aarhus C, Denmark

April 27, 2005

**Abstract**

Formal semantics is a topic of major importance in the study of programming languages. Its applications include documenting language design, establishing standards for implementations, reasoning about programs, and generating compilers.

These notes introduce *action semantics*, a recently-developed framework for formal semantics. The primary aim of action semantics is to allow *useful* semantic descriptions of *realistic* programming languages.

**Keywords:** semantic descriptions, action semantics, action notation, data notation, algebraic specifications, modules, unified algebras.

## 1 Introduction

*Denotational Semantics* [10] is a popular tool in theoretical studies of programming languages. The main reasons for its popularity seem to be that (i) the abstract mathematical nature of the higher-order functions used as denotations facilitates proving facts about them, and (ii) $\lambda$-notation allows a very concise and direct specification of such denotations. Unfortunately, it also seems that these same features have pragmatic consequences that make Denotational Semantics quite *unsuitable* for defining the semantics of realistic programming languages.

*Action Semantics* [6, 13, 7, 11] is essentially just Denotational Semantics, but denotations are taken to be so-called *actions*, rather than higher-order functions. Actions are abstract entities that have a more computational essence than functions: actions, when performed, process information *gradually*. Actions provide straightforward representations for the denotations of a wide range of programming constructs—including nondeterminism and concurrency, whose treatment in Denotational Semantics can be problematic.

The standard notation for actions, called *Action Notation*, is quite different from $\lambda$-notation. It enjoys simple algebraic laws, and has a clear operational interpretation. Its use ensures that action semantic descriptions have good *modifiability*, and that they scale up smoothly from small, illustrative examples to full, realistic programming languages. Moreover, the suggestiveness of the symbols used in Action Notation provides good *readability*.

Action Notation can be regarded as a basic *intermediate language*, specially designed for use in semantic descriptions. Then an action semantic description determines a *translation* from a programming language into Action Notation. The operational semantics of Action Notation induces an operational semantics for the described programming language, and the laws of Action Notation allow reasoning about semantic equivalence of programs, or parts of programs. The $\lambda$-notation used in Denotational Semantics, in comparison with Action Notation, should be regarded more as an abstract 'machine code' than as an intermediate language, although its mathematical theory compensates to some extent for its low level.

In this introduction to Action Semantics we start with a brief (and somewhat dry) presentation of a meta-notation for use in semantic descriptions. The meta-notation is based on *Unified Algebras* [8], which is a rather unconventional framework for algebraic specifications. We illustrate the use the meta-notation initially by specifying some standard abstract data types; then we see how to use it for presenting semantic descriptions. After that we introduce the main constructs of Action Notation. Finally, we illustrate the action semantic description of various programming constructs.

Please refer to [11] for a comprehensive presentation of Action Semantics, including the action semantic description of a substantial sublanguage of ADA.

**Pedagogical Remark:** These notes are intended to support a series of five 45-minute lectures, organized as follows:

1. Introduction.
   Meta-Notation.
   Data Notation (truth-values, numbers).

2. Data Notation (tuples, lists, trees).
   Semantic Descriptions.

3. Action Notation (basic, functional).
   Action Semantic Descriptions (expressions).

4. Action Notation (declarative, abstractions, imperative).
   Action Semantic Descriptions (declarations, statements).

5. Action Notation (foundations, extensions).
   Action Semantic Descriptions (procedures), if time available.
   Conclusion.

Notice that the presentation of Action Notation is best interleaved with illustrations of its use in lectures, although for ease of reference, its explanation in these notes is kept separate. By the way, if you would like to get an impression of the appearance of action semantic descriptions before we start on the preliminaries, you should look at Section 6.1 and some of Section 6.3.

# 2 Meta-Notation

Meta-notation is for specifying formal notation: what symbols are used, how they may be put together, and their intended interpretation.

Our meta-notation here supports a *unified* treatment of sorts and individuals: an individual is treated as a special case of a sort. Thus operations can be applied to sorts as well as individuals. A vacuous sort represents the lack of an individual, i.e., the 'undefined' result of a partial operation. Sorts may be related by inclusion, and sort equality is just mutual inclusion. But a sort is not determined just by the set of individuals that it includes, i.e., its 'extension': it also has an 'intension', stemming from the way it is expressed. For example, the sort of those natural numbers that are in the range of the successor operation has a different intension from the sort of those that have a well-defined reciprocal, even though their sets of individuals are the same.

The meta-notation provides Horn clauses and constraints—explained below—for specifying the intended interpretation of symbols. Specifications may be divided into mutually-dependent and nested modules, which may be presented incrementally. Our meta-notation has been designed especially for unobtrusive use in action semantic descriptions. Its merits relative to conventional specification languages such as OBJ3 are discussed in [9].

The *vocabulary* of the meta-notation consists of (constant and operation) symbols, variables, titles, and special marks. *Symbols* are of two forms: quoted or unquoted. Quoted symbols always stand for constants (characters or strings). In unquoted symbols the character _ indicates argument positions. Unquoted symbols are written here in this sans-serif font. An operation symbol is classified as an infix when it both starts and ends with a _ , and as a prefix or postfix when it only ends, respectively starts, with a _ . There are three built-in symbols: nothing, _ | _ , and _ & _ . *Variables* are sequences of letters, here written in *this italic font*, optionally followed by primes ′ and/or a numerical subscript. *Titles* are sequences of words, here Capitalized and written in **This Bold Font**.

A pair of grouping parentheses ( ) may be replaced by a vertical rule to the left of the grouped material. Horizontal rules separate formal specification from informal comments. Reference numbers for parts of specifications have no formal significance.

A *sentence* is essentially a Horn clause involving formulae that assert equality, sort inclusion, or individual inclusion between the values of terms. The variables occurring in the terms range over all values, not only over individuals.

*Terms* consist of constant symbols, variables, and applications of operation symbols to subterms. We use *mixfix* notation, writing the application of a operation symbol $S_0\_\ldots\_S_n$ to terms $T_1, \ldots, T_n$ as $S_0T_1\ldots T_nS_n$. Infixes have weaker precedence than prefixes, which themselves have weaker precedence than postfixes. Grouping parentheses ( ) may be inserted for further disambiguation. Parentheses may also be omitted when alternative ways of reinserting them lead to the same interpretation. E.g., the operation _ | _ is associative, so we may write $x \mid y \mid z$ without disambiguating the grouping.

The value of the constant nothing is a vacuous sort, included in all other sorts. All operations map sorts to sorts, preserving sort inclusion. _ | _ is sort union and _ & _ is sort intersection; they are the join and meet of the sort lattice.

There are three kinds of basic *formula*: '$T_1 = T_2$' asserts that the values of the terms $T_1$ and $T_2$ are the same (individuals or sorts). '$T_1 \leq T_2$' asserts that the value of the term $T_1$ is a subsort of that of the term $T_2$. Sort inclusion is a partial order. '$T_1 : T_2$' asserts that the value of the term $T_1$ is an individual included in the (sort) value of the term $T_2$. Thus '$T : T$' merely asserts that the value of $T$ is

an individual.

'$F_1$ ; ... ; $F_n$' is the conjunction of the formulae $F_1, \ldots, F_n$. A (generalized Horn) *clause* '$F_1$; ... ; $F_m \Rightarrow C_1$; ... ; $C_n$' asserts that whenever all the antecedent formulae $F_i$ hold, so do all the consequent clauses (or formulae) $C_j$. Note that clauses cannot be nested to the left of $\Rightarrow$ .

We may restrict the interpretation of a variable $V$ to individuals of some sort $T$ in a clause $C$ by specifying '$V{:}T \Rightarrow C$'. Alternatively we may simply replace some occurrence of $V$ as an argument in $C$ by '$V{:}T$'. We restrict $V$ to subsorts of $T$ by writing '$V \leq T$' instead of '$V{:}T$'.

The mark $\Box$ (read as 'filled in later') in a term abbreviates the other side of the enclosing equation. Thus $T_2 = T_1 \mid \Box$ specifies the same as $T_2 = T_1 \mid T_2$ (which is equivalent to $T_2 \geq T_1$). The mark *disjoint* following an equation $T = T_1 \mid \ldots \mid T_n$ abbreviates the equations $T_i \,\&\, T_j = \mathsf{nothing}$, for $1 \leq i < j \leq n$. Similarly, the mark *individual* abbreviates formulae asserting that the $T_i$ are disjoint individuals.

A *functionality* clause '$S :: T_1, \ldots, T_n \rightarrow T$' is an abbreviation[1] which specifies that the value of any application of $S$ is included in $T$ whenever the values of the argument terms are included in the $T_i$. Note that it does *not* indicate whether the value might be an individual, a proper sort, or a vacuous sort.

Such a functionality may be augmented by the following *attributes* (defined rigorously in [11, Appendix D]): *strict*: the value is nothing when any argument is nothing; *linear*: the value on a union of two sorts is the union of the values on each sort separately, and similarly for intersections; *total*: the value is an individual when all arguments are individuals—moreover, $S$ is *strict* and *linear*; *partial*: as for *total*, except that the value is either an individual or a vacuous sort when the arguments are individuals.

When $S$ is binary, we may use the following attributes, following OBJ3: *associative*, *commutative*, *idempotent*, and *unit is* $T'$. These attributes have a similar meaning when $S$ is unary and the argument sort is a tuple sort, such as $T^+$ or $(T_1, T_2)$. (See Section 3 for the notation for tuples, which is not regarded as a part of the meta-notation itself.)

In all cases, the attributes only apply when all arguments are included in the sorts specified in the functionality. For instance, consider:

```
product _ ::
    (number, number) → number (total, associative, commutative, unit is 1) ,
    (matrix, matrix) → matrix (partial, associative)
```

which also illustrates how two or more functionalities for the same symbol can be specified together.

It is straightforward to translate ordinary many-sorted algebraic specifications into our meta-notation, using functionalities and attributes; similarly for order-sorted specifications [2] written in OBJ3 [4]. Sorted signatures translate to unsorted signatures together with axioms; sorted axioms translate to conditional unsorted axioms.

Let us now proceed to compound specifications. A *modular specification* $S$ is of the form '$B\ M_1 \ldots M_n$', where $B$ is a basic specification, and the $M_i$ are modules.

[1] Not much of an abbreviation: it expands to '$S\ (T_1, \ldots, T_n) \leq T$'. The monotonicity of all operations ensures the intended interpretation.

Either $B$ or the $M_i$ may be absent. $B$ is inherited by all the $M_i$. Each symbol in a specification stands for the same value or operation throughout—except for symbols introduced 'privately'. All the symbols (but not the variables) used in a module have to be explicitly introduced: either in the module itself, or in an outer basic specification, or in a referenced module.

A *basic specification* $B$ may introduce symbols, assert sentences, and impose constraints on subspecifications. The meta-notation for basic specifications is as follows.

'**introduces:** $S_1, \ldots, S_n$ .' introduces the indicated symbols, which stand for constants and/or operations. Also the lesser-used '**privately introduces:** $S_1, \ldots, S_n$ .' introduces the indicated symbols, but here the enclosing module translates them to 'new' symbols, so that they cannot clash with symbols specified in other modules.

'$S$ .' asserts that the sentence $S$ holds for any assignment of values to the variables that occur in it. '$B_1 \ldots B_n$' is the union of the basic specifications $B_1, \ldots, B_n$.

'**includes:** $R_1, \ldots, R_n$ .' specifies the same as all the modules indicated by the references $R_i$. '**needs:** $R_1, \ldots, R_n$ .' is similar to '**includes:** $R_1, \ldots, R_n$ .', except that it is not transitive: symbols introduced in the modules referenced by the $R_i$ are not regarded as being automatically available for use in modules that reference the enclosing module. '**grammar:** $S$' augments the basic specification $S$ with standard specifications of strings and trees from Data Notation, and with the explicit introduction of each constant symbol that occurs as a nonterminal, i.e., as the left-hand-side of an equation in $S$.

'**closed** .' specifies the *constraint* that the enclosing module is to have a 'standard' (i.e., initial) interpretation. This means that it must be possible, using the specified symbols, to express every *individual* that is included in some expressible sort ('no junk'), and moreover that terms have equal/included/individual values only when that logically follows from the specified axioms ('no confusion'). '**closed except** $R_1, \ldots, R_n$ .' specifies a similar constraint, but leaves the submodules referenced by the $R_i$ open, so that they may be specialized in extensions of the specification. '**open** .' merely indicates that the module containing it has intentionally not been closed.

A *module* $M$ is of the form '$T$ $S$', where $T$ is a title (or a series of titles separated by /) that identifies the specification $S$. Modules may be specified incrementally, in any order. To show that a module is continuing an earlier specification with the same identification, the mark *(continued)* is appended to its title. Modules may also be nested, in which case an inner module inherits the basic specifications of all the enclosing modules, together with the series of titles that identifies the immediately enclosing module. Parameterization of modules is rather implicit: unconstrained submodules, specified as '**open** .', can always be specialized, which provides a simple yet expressive form of instantiation.

A series of titles '$T_1/\ldots/T_n$' refers to a module, together with all that its submodules specify. '$T$ $(S'_1$ *for* $S_1, \ldots, S'_n$ *for* $S_n)$' refers to the same module as the titles $T$, but with all the symbols $S_i$ translated to $S'_i$. Identity translations '$S_i$ *for* $S_i$' may be abbreviated to $S_i$, as in '$T$ $(S_1, \ldots, S_n)$' which merely indicates that the module referenced by $T$ specifies at least all the symbols $S_1, \ldots, S_n$.

In subsequent sections we see how to use this meta-notation for specifying data

notation (i.e., abstract data types), abstract syntax, semantic functions, and laws of action notation. [11] provides further examples of use, as well as foundations.

# 3 Data Notation

Various sorts of data are needed for the semantics of general-purpose high-level programming languages, not only 'mathematical' values such as numbers and lists, but also abstract entities of computational origins such as variables, files, procedures, objects, modules, and so on.

It would be futile to try to provide standard notation for all possible sorts of data. Apart from the excessive amount of notation that would be needed, future programming languages may involve sorts of data previously unconceived. Action Semantics provides the following sorts of data, which—together with appropriate operations—comprise our basic *Data Notation*:

**Truth Values:** the usual 'Booleans'. Predicates are represented as total truth-valued operations.

**Numbers:** unbounded exact rational numbers. Restriction to bounded numbers can easily be expressed using sort intersection. A loosely-specified sort of 'approximations' can be specialized to represent the usual types of implemented 'real' numbers (fixed-point, floating-point).

**Characters:** an unspecified character set. The ASCII character set is provided too.

**Lists:** ordered, possibly nested, collections of arbitrary items.

**Strings:** unbounded lists of characters.

**Trees:** nested lists. Trees with characters as leaves are used as syntactic entities.

**Sets:** unordered, possibly nested, collections of arbitrary (but distinguishable) elements.

**Maps:** unordered collections of arbitrary items, indexed by distinguishable elements.

**Tuples:** ordered single-level collections of arbitrary components. Single components are 1-tuples. We represent operations with varying numbers of arguments as unary operations on tuples. For example, list of _ makes a list from a tuple of items.

Lists, sets, maps and tuples are always *finite*, and their components are individuals (not vacuous or proper sorts). Infinite and 'lazy' data can be represented by abstractions, which are explained in Section 5.

Apart from Data Notation, Action Semantics provides some further sorts of data, such as storage cells and abstractions. These are part of Action Notation, and described in Section 5. Any further sorts of data that are needed (for an action semantic description of a particular programming language) have to be specified algebraically, *ad hoc*.

[11, Appendix C] provides a complete (algebraic) specification of Data Notation. Here, we only have space for a few illustrative excerpts. By the way, the symbols of our notation are generally formed from highly suggestive, unabbreviated words, exploiting the occasional punctuation mark.

Our first example of Data Notation provides ordinary truth-values. But some of the operations are *polymorphic*! For instance, if true then $x$ else $y$ is always $x$, regardless of whether $x$ is a truth-value or some other entity.

## 3.1 Truth-Values/Basics

**introduces:** truth-value , true , false .

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

**closed** .

The constraint **closed** ensures that true and false are the only individuals of sort truth-value, and that they are not the same individual. This constraint must be observed in every module that refers to **Truth-Values/Basics**.

## 3.2 Truth-Values/Specifics

**introduces:** if _ then _ else _ , when _ then _ , there is _ , not _ .

**includes:** **Basics** .

'**Basics**' is a relative reference, abbreviating '**Truth-Values/Basics**'.

(1) if _ then _ else _ :: truth-value, $x$, $y \to x$ | $y$ .

(2) when _ then _ :: truth-value, $x \to x$ (*partial*) .

(3) there is _ :: $x \to$ true (*total*) .

(4) not _ :: truth-value $\to$ truth-value (*total*) .

Now the details:

(5) (if $t$:truth-value then $x$ else $y$) = when $t$ then $x$ | when not $t$ then $y$ .

(6) (when true then $x$) = $x$ . (when false then $x$) = nothing .

(7) (there is $x$:$x$) = true . (there is nothing) = nothing .

(8) (not true) = false . (not not $t$:truth-value) = $t$ .

Data Notation also provides conjunction all _ and disjunction any _ on tuples of truth-values (as well as their restrictions to pairs: both _ and either _). We omit their specification here, as we have not yet introduced our notation for tuples.

Notice that that **Truth-Values/Specifics** observes the constraint imposed by **Truth-Values/Basics**: any individual of sort truth-value expressible using the introduced operation symbols is equated to true or to false—but not to both of them!

Let us next consider the following specification of natural numbers. The intended interpretation of the introduced symbols is fully specified, and corresponds closely to the familiar standard model of natural numbers.

## 3.3 Numbers/Naturals/Basics

**introduces:** natural , positive-integer , successor _ , 0 , 1 , 2 .

(1) natural = 0 | positive-integer (*disjoint*) .

(2) successor _ :: natural → positive-integer (*total*) .

(3) 0 : natural . 1 = successor 0 . 2 = successor 1 .

**closed** .

Please draw a (Hasse) diagram of the lattice formed by those sorts expressible by terms in **Naturals/Basics**. Are the individuals all just above the value of nothing? Is there any relation between the values of positive-integer and successor (natural)? (See the appendix for answers to such questions.)

Further operations on natural numbers are specified later in this section, after we specify tuples, which play a major rôle in Data Notation. Tupling is associative, like string concatenation, so tuples cannot be nested directly.

## 3.4 Tuples

### 3.4.1 Generics

**introduces:** component .
**open** .

### 3.4.2 Basics

**introduces:** tuple , ( ) , ( _ , _ ) , _? , _* , _+ .

The symbol ( _ , _ ) is unusual in that it incorporates its own parentheses. These can be omitted when it is applied iteratively, because it is associative, as specified below.

**includes:** **Generics** .
(1) tuple = ( ) | component | (component+, component+) (*disjoint*) .
(2) ( ) : tuple .
(3) ( _ , _ ) :: tuple, tuple → tuple (*total*, *associative*, *unit is* ( )) .
(4) _? , _* , _+ :: tuple → tuple .
(5) $x^? =$ ( ) | $x$ . $x^* =$ ( ) | $x^+$ . $x^+ = x$ | $(x^+, x^+)$ .
**closed** **except** **Generics** .

We have not specified any attributes at all for the iteration operator _*. Clearly $x^*$ is generally a proper sort, not an individual, and never vacuous, so we shouldn't specify *strict*, *total*, or *partial*. But how about *linear*?

The specification of tuples is generic, because the sort component has been left open. There are two ways of instantiating tuples to allow, say, natural numbers as components: syntactically, by including the translated specification **Tuples** (natural *for* component, natural-tuple *for* tuple); or semantically, by including **Tuples** unchanged and specifying natural ≤ component as an axiom. The semantic approach is preferable, as it avoids the need for introducing new (sort) symbols. By the way, ( ) is the empty tuple of *any* sort. Here are some further operations on tuples:

### 3.4.3 Specifics

**introduces:** _ - , count _ , component#_ _ , distinct _ .
**includes:** **Basics**.
(1) _ - :: tuple, natural → tuple .
(2) count _ :: tuple → natural (*total*) .
(3) component#_ _ :: positive-integer, tuple → component (*partial*) .
(4) distinct _ :: (component+, component+) → truth-value (*partial*, *commutative*) .
(5) (1) $x^0 =$ ( ) .
(2) $x^{\text{successor } n:\text{natural}} = (x, x^n)$ .
(6) (1) count ( ) = 0 .
(2) count ($c$:component, $t$:tuple) = successor count $t$ .

(7) (1) component#($i$:positive-integer) ( ) = nothing .

(2) component#(1) ($c$:component, $t$:tuple) = $c$ .

(3) component#(successor $i$:positive-integer) ($c$:component, $t$:tuple) =
component#($i$) $t$ .

(8) (1) distinct ($x$:component, $y$:component) = not ($x$ is $y$) .

(2) distinct ($x$:component$^+$, $y$:component, $z$:component$^+$) =
all (distinct ($x$, $y$), distinct ($x$, $z$), distinct ($y$, $z$)) .

Let us now continue our specification of natural numbers by specifying sums and products, using tuples. The attribute *associative* for a unary operation f _ (on tuples) specifies that f ($x$, $y$, $z$) is equal to f (f ($x$, $y$), $z$) and to f ($x$, f ($y$, $z$)). Similarly *unit is* $u$ equates
f ($x$, $u$) to $x$ and f ( ) to $u$.

## 3.5 Numbers/Naturals/Specifics

**introduces:** sum _ , product _ .
**includes:** **Basics** .
**needs:** **Tuples/Basics** . natural $\leq$ component .

(1) sum _ :: natural$^*$ $\rightarrow$ natural (*total*, *associative*, *commutative*, *unit is* 0) .

(2) product _ :: natural$^*$ $\rightarrow$ natural (*total*, *associative*, *commutative*, *unit is* 1).

(3) sum ($n$:natural, 1) = successor $n$ .

(4) product ($n$:natural, 0) = 0 .

(5) product ($m$:natural, successor $n$:natural) = sum ($m$, product ($m$, $n$)) .

In fact sum _ and product _ are fully defined on natural$^*$ by the above specification. Can you see how to use the attributes, together with axiom (3), to convert any term of the form sum (successor$^m$ 0, successor$^n$ 0) to the term successor$^{m+n}$ 0 ?

Although the above extension does not introduce any new *individuals*, it allows plenty of new *subsorts* of sort natural to be expressed! Constraints only concern individuals, so our extension doesn't conflict with the constraint on **Naturals/Basics**. Extensions are also allowed to equate sorts that were previously unrelated in a constrained module—so long as this doesn't affect individuals.

It is an amusing exercise to investigate which sorts of individuals are expressible by terms in **Naturals/Basics** and **Naturals/Specifics**. For instance, are all cofinite sorts of natural numbers expressible? How about a sort including all even numbers?

Our final example here is the entire specification of generic lists. The specifications of generic sets and maps would be similar.

## 3.6 Lists

### 3.6.1 Generics

**introduces:** nonlist-item .
**open** .

### 3.6.2 Basics

**introduces:** list , item , list of _ .
**includes:** **Generics** .
**needs:** **Tuples/Basics** . list $\leq$ component .
(1) list = list of item$^*$ .
(2) item = nonlist-item | list $(disjoint)$ .
(3) list of _ :: item$^*$ $\rightarrow$ list $(total)$ .
**closed except Generics** .

### 3.6.3 Specifics

**introduces:** [ _ ] _ , items _ , head _ , tail _ , empty-list , concatenation _ .
**includes:** **Basics** .
**needs:** **Tuples/Basics** .
(1) [ _ ] _ :: item, list $\rightarrow$ list .
(2) items _ :: list $\rightarrow$ item$^*$ $(total)$ .
(3) head _ :: list $\rightarrow$ item $(partial)$ .
(4) tail _ :: list $\rightarrow$ list $(partial)$ .
(5) empty-list : list .
(6) concatenation _ :: list$^*$ $\rightarrow$ list ($total$, $associative$, $unit\ is$ empty-list) .
(7) [ $i\leq$item ] $l\leq$list = $l$ & list of $i^*$ .
(8) $l$ = list of $i$ $\Rightarrow$ items $l$:list = $i$ .
(9) head list of ($i$:item, $i'$:item$^*$) = $i$ . tail list of ($i$:item, $i'$:item$^*$) = list of $i'$ .
(10) empty-list = list of ( ) . concatenation ($l_1$:list, $l_2$:list) = list of (items $l_1$, items $l_2$) .

We can instantiate generic lists in the same way as tuples. Notice that when natural $\leq$ component, we automatically get [positive-integer] list $\leq$ [natural] list, by monotonicity.

The following module provides strings and syntax-trees:

## 3.7 Trees/Syntax

**introduces:** string , syntax-tree .
**needs:** **Characters/Generics, Lists** . character $\leq$ nonlist-item .
(1) string = [character] list .
(2) syntax-tree = string | [syntax-tree] list .

Note that this only gives *finite* trees: the above axiom is *not* a domain equation, and we demand only monotonicity, not continuity, from operations.

Data Notation introduces abbreviations $[\![\ \_\ ]\!]$, $[\![\ \_\ \_\ ]\!]$, ..., for constructing nodes of trees, such that $[\![\ t_1\ \ldots\ t_n\ ]\!]$ is list of $(t_1, \ldots, t_n)$, for $n > 0$—where the $t_i$ may be *tuples* of trees. Similarly, the abbreviations $\langle\ \_\ \ldots\ \_\ \rangle$ allow the omission of commas in tuples. The use of these abbreviations is illustrated in the next section.

# 4 Semantic Descriptions

This section explains how to specify abstract syntax and semantic functions using our meta-notation. You are assumed to be familiar with the general idea of abstract syntax, and its relation to concrete syntax (otherwise see, e.g., [10, 11]). You probably also know that a semantic function is a map that takes each abstract syntactic entity to a semantic entity called its *denotation*, which represents its contribution to program behaviour. The map is required to be *compositional*, in that the denotation of each compound entity is determined by the denotations of its components—not by their form.

Compositionality is the basic feature that distinguishes denotational (and action) semantics from operational and axiomatic semantics. It can be formulated algebraically [3]: abstract syntax is the initial $\Sigma$-algebra, and semantic functions are the components of the unique $\Sigma$-homomorphism from abstract syntax to a target $\Sigma$-algebra. Thus it is sufficient to define just the target algebra, leaving the semantic functions implicit. In practice, however, the direct inductive definition of semantic functions by *semantic equations*, as in ordinary Denotational Semantics, tends to be more perspicuous than the definition of a target algebra.

We illustrate the specification of abstract syntax and semantic functions with a simple language of binary numerals. Let the concrete syntax be given by the following grammar, which is written in an extended BNF variant that is commonly used in programming language reference manuals:

```
binary = "2" "#" bits "#" .
bits   = bit { bit } .
bit    = "0" | "1" .
```

The terminal symbols are "0", "1", "2", and "#". The somewhat peculiar notation '{...}' indicates zero or more repetitions of '...'.

As semanticists, we may choose any abstract syntax that we like—provided that we are prepared to explain how concrete derivation trees are supposed to be mapped to abstract syntactic entities! Consider the following specification, written in our meta-notation:

## 4.1 Abstract Syntax

**grammar:**

(1) Binary = ⟦ "2" "#" Bits "#" ⟧ .

(2) Bits = Bit | ⟦ Bits Bit ⟧ .

(3) Bit = "0" | "1" .

**closed** .

The meta-notation '**grammar:**' merely has the effect of introducing the constant symbols Binary, Bits, and Bit, together with standard Data Notation for strings "..." and trees ⟦...⟧. This makes the rest of the specification well-formed. By the way, let us reserve Capitalized symbols for use with abstract syntax, and use lower case elsewhere. This convention removes the danger of a clash between syntactic

constants (which should be closely related to the nonterminals of the given concrete syntax, for the sake of perspicuity) and the symbols of the standard Action Notation.

Each equation above defines the value of a constant to be a sort of tree. The sort ⟦ "2" "#" Bits "#" ⟧ includes just those individual trees that have the string "2" as first component, "#" as second component, an arbitrary tree of sort Bits as third component, and "#" again as fourth and last component. (The "#"s could just as well have been dropped, but they do make the intended mapping from concrete to abstract syntax more obvious.) Similarly the sort ⟦ Bits Bit ⟧ includes just trees that have two components, the first a tree of sort Bits, the second a tree—actually a string—of sort Bit. Thus Bits is the union of that sort with the sort Bit, which is itself the union of the *individual* sorts "0" and "1".

Apart from the presence of the tree constructors ⟦...⟧, our specification looks like an ordinary context-free grammar. But it is entirely algebraic! Each equation is an algebraic axiom—thanks to our treatment of sorts as values. The specified sorts of trees are undisturbed when terms are replaced by equals, for instance Bit can be replaced by ("0" | "1") in the second equation above (making the third equation redundant).

A further significant feature of our specification is that we have Bit ≤ Bits, so our syntax is *order-sorted*. This turns out to be rather useful when specifying semantics.

We now specify the expected semantics of binary numerals. We treat semantic functions as ordinary operations, writing the semantic equations that define them as algebraic axioms in our meta-notation.

## 4.2 Semantic Functions

**needs:** **Abstract Syntax, Semantic Entities** .

**introduces:** the value of _ , the binary value of _ .

- the value of _ :: Binary → natural .

(1) the value of ⟦ "2" "#" $B$:Bits "#" ⟧ = the binary value of $B$ .

The functionality of the semantic function merely provides a concise summary of the sort of denotation to be defined for a particular sort of syntactic entity. The semantic equation is actually an abbreviation for a clause with antecedent '$B$:Bits'.

- the binary value of _ :: Bits → natural .

(2) the binary value of "0" = 0 .

(3) the binary value of "1" = 1 .

(4) the binary value of ⟦ $B$:Bits $B'$:Bit ⟧ =
sum (product (2, the binary value of $B$), the binary value of $B'$) .

It is easy to check that these equations define the value of $B$ to be the expected natural number for every individual $B$ of sort Binary. (The attribute *total* would extend the definition strictly and linearly to all subsorts of Binary—as would *partial*. But here we are only interested in applying semantic functions to individuals, so let's not bother with such details.) A formal proof relies on the constraint **closed** on

the abstract syntax, which restricts individual trees to being finite and prevents different-looking trees or strings from being equal: 'no junk' and 'no confusion', following the usual explanation of initiality.

Notice that the same symbol was used for the semantic functions on Bits and its subsort Bit. Had two different symbols been used, we would have needed a rather uninformative semantic equation to relate their values on Bit. On the other hand, different symbols were used for the semantic functions on Binary and Bits. That merely facilitates adding other kinds of numerals to the described language without any change to the given semantic equations.

Our semantic description of binary numerals isn't quite complete, as it refers to a module **Semantic Entities**, which hasn't yet been specified. But Data Notation already provides natural numbers, so all we need to do is specify:

## 4.3 Semantic Entities

**includes: Data Notation/Numbers/Naturals**.

Let us conclude this section by reconsidering our choice of abstract syntax. You may have noticed that our abstract syntax grammar for Bits used *left* recursion. Couldn't we have chosen right recursion—or even Bits = Bit | ⟦ Bits Bits ⟧ instead? No, not if we want the denotation of $B$ of sort Bits to be the expected natural number! For when $B$ is more than just a single bit, the binary value of ⟦ "1" $B$ ⟧ is determined not only by the binary value of $B$, but also by its *length*! Thus a compositional semantics for such an abstract syntax would require the denotation of $B$ to be the *pair* of its value and length. Quite often, choice of abstract syntax is not a trivial matter, and one has to compromise between the conflicting aims of keeping close to concrete syntax and allowing simple denotations.

In the case of binary numerals, there is another possibility: to use trees with *arbitrary* (finite) branching. This involves the use of the notation for sorts of tuples, as follows.

## 4.4 Lexical/Abstract Syntax

**grammar:**

(1) Binary = ⟦ '2' '#' Bit$^+$ '#' ⟧ .

(2) Bit = '0' | '1' .

**closed** .

We take the opportunity to illustrate the use of characters, instead of strings, as terminal symbols. A tree whose direct components are all characters is just a string, for instance ⟦ '2' '#' '1' '1' '0' '#' ⟧ (of sort Binary here) is the same as "2#110#". Syntactic entities that correspond to *lexemes* (the result of concrete lexical analysis) can generally be represented as strings and specified in this way.

The semantic functions are much as before, except that the inductiveness of the definition now comes from the division of a tuple into a nonempty tuple of bits and a single bit. In fact the use of tuples instead of nesting leaves it open whether

semantic functions are defined inductively from the left or from the right. That flexibility would be useful here if we were to add binary *fractions* to our example.

### 4.5 Lexical/Semantic Functions

**needs:** **Abstract Syntax, Semantic Entities** .

**introduces:** the value of _ , the binary value of _ .

- the value of _ :: Binary → natural .

(1) the value of ⟦ '2' '#' $B$:Bit$^+$ '#' ⟧ = the binary value of $B$ .

- the binary value of _ :: Bit$^+$ → natural .

(2) the binary value of '0' = 0 .

(3) the binary value of '1' = 1 .

(4) the binary value of ⟨ $B$:Bit$^+$ $B'$:Bit ⟩ =
sum (product (2, the binary value of $B$), the binary value of $B'$) .

By the way, also the VDM approach to (denotational) semantics [1] advocates the use of tuples in abstract syntax. Its basic notation for abstract syntax is, however, rather less suggestive than that used here. An additional disadvantage is that it allows sets and maps of components, and the resulting inherent lack of order of branches makes it uncertain that semantic functions are well-defined.

## 5 Action Notation

Action Notation is used for expressing semantic entities that represent the implementation-independent behaviour of programs, and the contributions that parts of programs make to overall behaviour. There are three kinds of semantic entity: actions, data, and dependent data. The main kind is, of course, actions; data and dependent data are auxiliary. Let us first consider the general nature of these entities, before looking at notational details.

*Actions* are essentially *computational* entities, directly representing information processing behaviour and reflecting the gradual, step-wise nature of computation. Actions can be *performed* so as to process information. A performance of an action, which may be part of an enclosing action, either *completes*, corresponding to normal termination (the performance of the enclosing action proceeds normally); or *escapes*, corresponding to exceptional termination (the enclosing action is skipped until the escape is trapped); or *fails*, corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or *diverges*, corresponding to nontermination (the enclosing action also diverges).

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

The information processed by action performance may be classified as follows: *transient* information consists of tuples of data, corresponding to intermediate re-

sults; *scoped* information is bindings of tokens to data, corresponding to symbol tables; *stable* information is data stored in cells, corresponding to the values assigned to variables; and *permanent* information involves data irrevocably communicated between distributed actions.

The different kinds of information give rise to so-called *facets* of actions, focusing on the processing of at most one kind of information at a time: the *control* facet, processing independently of information; the *functional* facet, processing transient information (actions are given and give data); the *declarative* facet, processing scoped information (actions receive and produce bindings); the *imperative* facet, processing stable information (actions reserve and unreserve cells of storage, and change the data stored in cells); and the *communicative* facet, processing permanent information (actions send and receive messages, and offer contracts to agents).

The various facets of an action are independent. For instance, changing the data stored in a cell—or even unreserving the cell—does not affect any bindings. There are, however, some primitive *hybrid* actions, which provide finite representations of self-referential bindings by processing a mixture of scoped and stable information.

Transient information is given only on completion or escape, and scoped information is produced only on completion. In contrast, changes to stable information and extensions to permanent information are made *during* action performance, and are unaffected by subsequent divergence or failure.

*Dependent data* are entities that can be *evaluated* to yield data during action performance. The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the storage and buffer. Evaluation cannot affect the current information. Usually, evaluation yields an individual, but it may also yield a proper sort, or a vacuous sort that represents the undefined result of a partial operation.

Compound dependent data can be formed by the application of data operations to dependent data. The data yielded by evaluating a compound dependent data is the result of applying the operation to the data yielded by evaluating the operands. Thus data is a special case of dependent data, and always yields itself when evaluated.

The information processed by actions consists of items of *data*, organized in structures that give access to the individual items. Data can include various familiar mathematical entities, such as truth-values, numbers, characters, strings, lists, sets, and maps. It can also include entities such as tokens, cells, and agents, used for accessing other items, and some compound entities with data components, such as messages and contracts. Actions themselves are not data, but they can be incorporated in so-called abstractions, which are data. New kinds of data can be introduced *ad hoc*, for representing special pieces of information.

The rest of this section introduces various constructs of Action Notation. It specifies the functionality of each symbol, and sketches its intended interpretation. The level of detail should be sufficient for you to understand the examples of action semantics shown in Section 6. But you would need to study a more comprehensive exposition of Action Notation, including its formal operational semantics [11], before you could expect to be able write such examples yourself with full confidence. (Section 5.7 gives an impression of the foundations of Action Notation, in case you are curious about them.)

Action Notation consists mainly of action *primitives* and *combinators*. Each

primitive is concerned with one particular kind of information processing, and makes no contribution to the other kinds. In general, all dependent data in a primitive action is evaluated to data before performing the action. Each combinator, on the other hand, expresses a particular mixture of control flow and various kinds of information flow. Action Notation was designed to have sufficient primitives and combinators for expressing most common patterns of information processing straightforwardly, i.e., without simulating one kind of information processing by another.

The standard symbols used in Action Notation are formed from ordinary English *words*, written in lower case. In fact Action Notation mimics natural language: expressions standing for actions form imperative verb phrases involving conjunctions and adverbs, e.g., check it and then escape; whereas expressions standing for data form noun phrases, e.g., the items of the given list. Definite and indefinite articles can be exploited appropriately, e.g., choose a cell then reserve the given cell. (There are obvious similarities between the form of Action Notation and that of the programming languages COBOL and HYPERTALK, although the design of Action Notation was not directly influenced by either.)

These simple principles give a reasonably grammatical fragment of English, making sensibly-written specifications of actions quite readable—without sacrificing formality! Indentation, emphasized by vertical rules, is used to disambiguate the grouping of combinators, which are written infix; parentheses may also be used.

Compared to other formalisms, such as the $\lambda$-notation, Action Notation may appear to lack conciseness: each symbol consists of several letters, rather than a single sign. But the comparison should also take into account that each action combinator corresponds, in general, to a complex pattern of applications and abstractions in $\lambda$-notation. The increased length of each symbol seems to be far outweighed by its increased perspicuity.

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

Below, $A$, $A_1$, $A_2$ stand for arbitrary individual actions, i.e., individuals of sort act, whereas $D$, $D_1$, $D_2$ stand either for arbitrary individuals of dependent data, or for arbitrary subsorts of data. The combinators are generally *total* operations, but we don't bother to specify that. (Those who have read [7] should note that for technical simplicity, we no longer consider performing general *sorts* of actions, only individuals.)

## 5.1 Basic

(1) complete , escape , fail , commit , diverge , unfold : act .

(2) unfolding _ , indivisibly _ :: act → act .

(3) _ or _ :: act, act → act (*associative*, *commutative*, *idempotent*, *unit is* fail) .

(4) _ and _ :: act, act → act (*associative*, *unit is* complete) .

(5) _ and then _ :: act, act → act (*associative*, *unit is* complete) .

(6) _ trap _ :: act, act → act (*associative*, *unit is* escape) .

Basic action notation is primarily concerned with specifying flow of control. Performance of the primitive action complete simply terminates normally, whereas that of escape terminates abnormally, and that of fail aborts. Performance of diverge never terminates. In fact diverge is an abbreviation for unfolding unfold, where unfolding $A$ performs $A$ but whenever it reaches unfold, it performs $A$ instead.

The combined action $A_1$ or $A_2$ represents implementation-dependent choice between alternative actions. When the performance of the chosen action fails, however, the alternative is performed instead. Thus if $A_1$, $A_2$ are such that one or the other of them is always bound to fail, the choice is deterministic—in particular, $A_1$ or fail is equivalent to $A_1$. However, actions may commit their performance to the current alternative, so that a subsequent failure cannot be ignored (as with cut in PROLOG).

$A_1$ and $A_2$ represents implementation-dependent order of performance of the indivisible subactions of $A_1$, $A_2$. When these subactions cannot interfere with each other, it represents that their order of performance is simply irrelevant. A performance of $A_1$ and $A_2$ interleaves the steps of performances of $A_1$, $A_2$ (perhaps unfairly) until both have completed, or until one of them escapes or fails. indivisibly $A$ makes an indivisible action out of any non-diverging action.

$A_1$ and then $A_2$ represents normal, left to right, sequencing. It performs $A_2$ only when $A_1$ completes. Similarly, $A_1$ trap $A_2$ represents abnormal sequencing, performing $A_2$ only when $A_1$ escapes.

## 5.2 Functional

(1) give _ , choose _ :: dependent data → act .
(2) regive : act .
(3) check _ :: dependent truth-value → act .
(4) _ then _ :: act, act → act (*associative*, *unit is* regive) .
(5) given _ :: data → dependent data .
(6) given _#_ :: datum, natural → dependent datum .
(7) it : dependent datum .
(8) them : dependent data .
(9) datum ≤ component .
(10) data = datum* .
(11) a _ , an _ , the _ , of _ :: data → data .

Functional actions are primarily concerned with processing transient information. The sort of components of transient information is datum. It includes various sorts from Data Notation, and it may be extended to include other sorts, as required for particular purposes. data consists of tuples whose components are of sort datum.

The primitive action give $D$ completes, giving the data yielded by evaluating $D$, provided that this is an individual; it fails when $D$ yields nothing. choose $D$ generalizes give $D$ to make a choice between the individuals of a sort yielded by $D$. For instance, choose a natural always terminates, giving an arbitrary individual of

the sort natural. The action check $D$ requires $D$ to yield a truth-value; it completes when the value is true, otherwise it fails (without committing).

$A_1$ then $A_2$ represents normal functional composition of $A_1$, $A_2$. The data given by $A_1$ on completion are given to $A_2$. Otherwise, $A_1$ then $A_2$ is like $A_1$ and then $A_2$. The action regive propagates all the transient information that is given to it.

The dependent data given $D$ yields all the data given to its evaluation, provided that the entire tuple is of the data sort $D$. given $D\#n$ yields the $n$'th individual component of a given tuple, $n > 0$. it and them both yield the given data, but it insists that there should be only a single component. More generally, the dependent data the $D_1$ yielded by $D_2$ yields the same individual as $D_2$, when that is of sort $D_1$, otherwise nothing.

The dependent data 'a $D$' is equivalent to $D$; similarly for 'an $D$', 'the $D$' and 'of $D$'. This allows dependent data to be expressed rather naturally, if desired. Note that 'the' and 'of' are obligatory parts of some of the other operation symbols introduced below.

Also basic actions process transient information. The primitive actions complete and commit give the null tuple, but escape is analogous to regive and gives any data given to it. The combinators pass the given data on to their subactions, except that $A_1$ trap $A_2$ is analogous to $A_1$ then $A_2$, in that $A_2$ is given the data given (on escape) by $A_1$. The basic combinators and, and then collect up any data given by their subactions, concatenating it in the given order. Note in particular that $A_1$ and $A_2$ is *not* equivalent to $A_2$ and $A_1$ when both $A_1$, $A_2$ can complete giving non-null data.

## 5.3 Declarative

(1) bind _ to _ :: dependent token, dependent bindable → act .
(2) rebind : act .
(3) furthermore _ :: act → act .
(4) _ hence _ :: act, act → act (*associative*, *unit is* rebind) .
(5) _ before _ :: act, act → act (*associative*, *unit is* complete) .
(6) current bindings : dependent bindings .
(7) the _ bound to _ :: bindable, dependent token → dependent bindable .
(8) bindings ≥ [token to bindable] map .
(9) token ≤ distinct-datum .
(10) bindable ≤ datum .

Declarative actions are concerned with scoped information, which consists of bindings of tokens to data. The sorts token and bindable are open, to be specified by the user. Usually, tokens are strings of a particular form.

The primitive action bind $T$ to $D$ produces the binding of the token $T$ to the bindable individual yielded by $D$. It does *not* reproduce any of the received bindings! The action rebind, in contrast, merely reproduces all the received bindings, thereby extending their scope.

$A_1$ hence $A_2$ lets the bindings produced by $A_1$ be received by $A_2$, which limits their scope (unless they get reproduced by $A_2$). Thus it is analogous to functional

composition. The action furthermore $A$ produces the same bindings as $A$, together with any received bindings that $A$ doesn't override. The compound combination furthermore $A_1$ hence $A_2$ (recall that prefixes have higher precedence than infixes!) corresponds to block structure, with $A_1$ being the block head and $A_2$ the block body: received bindings are received by $A_2$ unless they are overridden by bindings produced by $A_1$. The action $A_1$ before $A_2$ is somewhat similar, but here the bindings produced by $A_1$, as well as those produced by $A_2$, are produced by the combination (although failure occurs if the bound tokens clash). This is also how $A_1$ and $A_2$ and the other basic and functional combinations treat produced bindings, but they all let the received bindings be received by their subactions without further ado—analogously to how $A_1$ and $A_2$ gives the given data to $A_1$, $A_2$.

There are further declarative combinators, not needed here, which correspond to hybrids of the above combinators with various basic and functional combinators. For instance, _ thence _ is a hybrid of _ then _ and _ hence _ . Nevertheless, there may still be mixtures of control, data, and binding flow that are difficult to express directly. To remedy this, the dependent data current bindings and the action produce $D$ are provided, so that bindings can be manipulated as data and subsequently produced.

Finally, the dependent data the $D$ bound to $T$ yields the current binding for the token $T$, provided that it is of sort $D$.

## 5.4 Abstractions

(1) enact _ :: dependent abstraction → act .
(2) application _ to _ :: dependent abstraction, dependent data → dependent abstraction .
(3) closure _ :: dependent abstraction → dependent abstraction .
(4) abstraction ≤ datum .
(5) abstraction of _ :: act → abstraction .

An abstraction is a datum that incorporates an action. In particular abstraction of $A$ incorporates the action $A$; but note that dependent data occurring in $A$ does not get evaluated when the abstraction is evaluated: it is left for evaluation during the performance of the action.

enact $D$ performs the action incorporated in the abstraction yielded by the dependent datum $D$. The performance of the incorporated action is not given any data, nor does it receive any bindings. However, data and/or bindings may have already been supplied to the incorporated action. For suppose that $D_1$ yields an abstraction that incorporates an action $A$. Then evaluation of the dependent datum application $D_1$ to $D_2$ yields an abstraction incorporating an action that gives the data yielded by $D_2$ to $A$. Similarly, the dependent datum closure $D_1$ yields an abstraction incorporating an action that lets the current (at evaluation-time) bindings be received by $A$.

The use of closure abstraction of $A$, instead of just abstraction of $A$, ensures so-called static bindings for abstractions that incorporate the action $A$. Then enact given abstraction performs $A$, letting it receive the bindings that were current when closure abstraction of $A$ was evaluated. The pattern enact application (given abstraction#1) to (rest given data) is useful for supplying parametric data to the abstraction, whereas enact closure (given abstraction) provides dynamic bindings (unless

static bindings were already supplied).

## 5.5 Imperative

(1) store _ in _ :: dependent storable, dependent cell → act .
(2) reserve _ , unreserve _ :: dependent cell → act .
(3) the _ stored in _ :: storable, dependent cell → dependent storable .
(4) storage = [cell to storable | uninitialized] map .
(5) cell ≤ distinct-datum .
(6) storable ≤ data .
(7) uninitialized : distinct-datum .

Imperative actions are concerned with stable information, which consists of the storage of data in cells. The sorts cell and storable are open. The organization of storage is usually implementation-dependent, so cell is left loosely specified, whereas storable is to be specified by the user.

The action store $D_1$ in $D_2$ changes the data stored in the cell yielded by $D_2$ to the storable datum yielded by $D_1$. It also commits the performance to the current alternative (otherwise implementations would have to be prepared to back-track to some previous storage upon failure). However, the cell concerned must have been previously reserved, using reserve $D$. There is usually no need to be specific about which cell is used—in fact Action Notation provides no operations for identifying particular cells! All one requires is a cell that is not currently reserved. This is provided by allocate $D$, where $D$ is a subsort of cell. It abbreviates a hybrid action:

```
indivisibly
| choose a [not in the mapped-set of the current storage] D then
| | reserve it and give it
```

where [not in $D_1$] $D_2$ is the subsort of $D_2$ that includes only those individuals that are not in the (finite) set $D_1$. Reserved cells can be made available for reuse by unreserve $D$.

The dependent datum the $D_1$ stored in $D_2$ yields the datum currently stored in the cell yielded by $D_2$, provided that it is of the sort $D_1$. It yields uninitialized between reserving the cell and storing something in it.

It is useful to be able to summarize the common features of some actions in terms of the various facets of their information processing. The following notation allows us to express sorts of actions on this basis, in a reasonably suggestive way.

## 5.6 Sorts

(1) [ _ ] act , perhaps _ :: act → act .
(2) bind , store ≤ act .
(3) [perhaps using _ ] act :: dependent data → act .
(4) dependent _ :: data → dependent data .

(5) [perhaps using _ ] dependent _ :: dependent data, data → dependent data .

[$A$] act restricts the sort of all actions act to those actions which, whenever performed, either fail or have an outcome in accordance with the action-sort $A$. Here, an action-sort is generated from complete, escape, diverge, give $D$, bind, and store using the combinators _ or _ , _ and _ , _ then _ , and perhaps _ , where perhaps $A$ is equivalent to $A$ or complete. For instance, we can express the sort [give a value or diverge] act, which excludes actions that complete without giving a value, escape, affect the storage, etc. The sort [perhaps escape and perhaps diverge and perhaps store] act allows arbitrary actions that neither give data nor produce bindings.

[perhaps using $D$] act restricts act on the basis of a sort $D$ of dependent data, generated from given $D'$, current bindings, and current storage using sort union _ | _ . Similarly [perhaps using $D$] dependent $D'$ restricts the sort dependent $D'$ of dependent data that always yield something included in the data sort $D'$, on the basis of the sort $D$.

## 5.7 Foundations

Lack of space precludes a detailed exposition of the the foundations of Action Notation. However, the following sketch may make it easier to understand the intended interpretation of the main action primitives and combinators, since it indicates the structure of the configurations, or states, of the operational semantics. For more details, see [11].

The operational semantics of Action Notation is specified formally as a *transition system* using the *structural* style advocated by Plotkin [14] and others. First we need the abstract syntax of actions, which is specified as follows:

### 5.7.1 Action Notation/Abstract Syntax

(1) Act = Simple-Act | ⟦ Prefix Act ⟧ | ⟦ Act Infix Act ⟧ .
(2) Prefix = "unfolding" | "indivisibly" | □ .
(3) Infix = "or" | "and" | "and then" | "then" | "trap" | "moreover" | "hence" | "thence" | □ .
(4) Simple-Act = "complete" | "escape" | "fail" | "commit" | "unfold" | ⟦ Simple-Prefix Dependent ⟧ | ⟦ "bind" Dependent "to" Dependent ⟧ | ⟦ "store" Dependent "in" Dependent ⟧ | □ .
(5) Simple-Prefix = "give" | "choose" | "produce" | "reserve" | "unreserve" | "enact" | □ .
(6) Dependent = □ .

Here is some of the specification of configurations, or *states*, written in our usual meta-notation:

### 5.7.2 Semantic Entities

(1) state = ( Acting, storage ) .

(2) info = ( data, bindings, storage ) .

(3) Acting = ⟨ "completed" data bindings ⟩ | ⟨ "escaped" data ⟩ | "failed" |
⟦ Prefix Acting ⟧ | ⟦ Acting Infix Acting ⟧ | ⟨ Act data$^{?}$ bindings$^{?}$ ⟩ | □ .

The difference between Act and Acting is that the latter allows data and bindings to be attached to subactions, for use when they get performed; moreover, Acting allows components that represent the information provided by terminated performances of subactions, which sometimes has to be combined with other information before being propagated.

Rather than specify a transition *relation*, we exploit the expressiveness of our usual meta-notation to specify a transition *function*, mapping each individual state to the entire *sort* of possible next states. This has some pragmatic advantages, for instance we can use an equation to specify that the sort of next states is a particular individual, when the transition happens to be deterministic.

### 5.7.3 Semantic Functions

- evaluated _ :: (Dependent, info) → data .
- stepped _ :: state → (state, commitment) .

(1) commitment = committing | uncommitted .

The commitment indicates whether a committing action has just been performed, in which case the current alternatives should be removed.

(2) stepped ("complete", $d$, $b$, $s$) =
("completed", ( ), empty-map, $s$, uncommitted).

The following examples indicate how we can specify transition dependencies, using Horn clauses instead of the conventional inference rules:

(3) evaluated $(D, d, b, s) = d'$ : data ⇒
stepped (⟦ "give" $D$:Dependent ⟧, $d$, $b$, $s$) =
("completed", $d'$, empty-map, $s$, uncommitted) .

(4) evaluated $(D, d, b, s) \geq d'$ : data ⇒
stepped (⟦ "choose" $D$:Dependent ⟧, $d$, $b$, $s$) ≥
("completed", $d'$, empty-map, $s$, uncommitted) .

(5) stepped $(A_1, s) \geq$ ("completed", $d$, $b$, $s'$, $c$) ⇒
stepped (⟦ $A_1$ "and" $A_2$ ⟧, $s$) ≥ (⟦ ("completed", $d$, $b$) "and" $A_2$ ⟧, $s'$, $c$) .

(6) stepped $(A_1, s) \geq$ ("failed", $s$, uncommitted) ⇒
stepped (⟦ $A_1$ "or" $A_2$ ⟧, $s$) ≥ ($A_2$, $s$, uncommitted) .

(7) stepped $(A_1, s) \geq (A_1', s', c'$:committing) ⇒
stepped (⟦ $A_1$ "or" $A_2$ ⟧, $s$) ≥ $(A_1', s', c')$ .

Of course, this is only a fragment of the complete specification given in [11], which is about 12 pages long (not counting explanatory comments).

## 5.8 Extensions

This section has introduced about 2/3 of the entire Action Notation. The omitted constructs are mainly concerned with asynchronously-communicating distributed systems of agents, and with the finite representation of self-referential bindings. They are introduced and exemplified in [11].

All that one has to do before using Action Notation in an action semantic description of a programming language is to specify the information that is to be processed by actions. This may involve specializing Data Notation and extending it with further data. The open sorts datum, token, bindable and storable should be specified. In fact the differences between bindable, storable, and some other sorts of data such as expression values are quite revealing about the essence of the language being described [16].

Note that one may introduce formal abbreviations for commonly-occurring patterns of notation that correspond to language-dependent concepts. For instance, one may specify an action assign $D_1$ to $D_2$ as a generalization of store $D_1$ in $D_2$ to arbitrary variables that may not be represented by single cells of storage.

The full Action Notation supports the specification of many important programming concepts. But it does not claim to be universal—except in the sense of Turing-completeness, of course. Unsupported concepts include (general) continuations, and real time. Continuations are not supported because traps and escapes are adequate to deal with the semantics of labels and goto's (more or less as in VDM [1]), and because they would somewhat complicate the operational semantics of Action Notation. This unfortunately seems to preclude a simple action semantic description of SCHEME. Real time could be added without too much trouble regarding the operational semantics, but this might invalidate some of the laws of Action Notation. The precise limits of the applicability of Action Notation remain to be seen.

The next section gives some examples of the use of Action Notation.

# 6 Action Semantic Descriptions

The preceding sections introduced all that we need for specifying action semantic descriptions of programming languages. We now have a convenient meta-notation for specifying abstract syntax, semantic entities, and semantic functions; and we have Action Notation, which provides semantic entities called actions that have a rather straightforward operational interpretation—together with suggestive symbols for them.

This section gives some examples of action semantic descriptions. The main purpose of the examples is to show how fundamental concepts of programming languages (sequential computation, scope rules, local variables, etc.) are reflected by the use of Action Notation. Our analysis of programming languages into fundamental concepts is essentially the same as that used in Denotational Semantics, following the insight of Christopher Strachey and his colleagues [15].

The programming constructs dealt with in the examples below are, in general, *simplified* versions of constructs to be found in conventional high-level programming languages. The agglomeration of the exemplified constructs would not make a particularly elegant and/or practical programming language. (In fact the examples are essentially the same constructs as in [12], and a subset of those given in [10],

so as to facilitate comparison between Action Semantics and two different styles of Denotational Semantics.)

Section 6.1 specifies denotations for arithmetical and Boolean expressions, using basic and functional actions. Section 6.2 shows how to specify denotations for constant declarations, including function abstractions, using declarative actions. Then Section 6.3 deals with statements and variable declarations, using imperative actions. Finally, Section 6.4 describes procedures with various modes of parameter evaluation. The abstract syntax chosen for the examples is easy to relate to the constructs of high level programming languages such as PASCAL and STANDARD ML.

A notable feature of our examples is that the introduction of the later constructs does *not* require changes to the already-given description of the earlier constructs. This phenomenon, which we call *extensibility*, has often been observed during the development of action semantic descriptions: one can start by describing a simple sublanguage, without regard to the rest of the language, and retain its description *unchanged* when extending to the full language.

It seems that this feature is unique to Action Semantics. It is due to the *polymorphism* of the combinators of Action Notation: the functional composition $A_1$ then $A_2$ remains a valid action when $A_1$, $A_2$ change the storage or communicate, for instance. Extensibility is definitely not a feature of conventional Denotational Semantics [10] where the use of the $\lambda$-notation makes semantic equations very sensitive to the detailed representation of denotations as higher-order functions. Even the use of monads in Denotational Semantics [5] does not provide extensibility approaching that of Action Semantics, it seems.

One can make a compromise between Denotational and Action Semantics by using action combinators in semantic equations, and defining them as functions on domains [12]. When new constructs are added to the described language, the original semantic equations generally remain valid, although the definitions of the combinators may have to be rewritten. Essentially, this way one is providing denotational models for increasing subsets of Action Notation, instead of exploiting the operational semantics of the entire Action Notation [11] as in pure Action Semantics. But it is difficult, if not impossible, to give a domain-based denotational model for the full Action Notation, with semantic equivalence for actions satisfying all the intended laws. This is because of features such as nondeterministic interleaving and concurrency, whose treatment in Denotational Semantics is rather unsatisfactory. (It uses so-called resumptions, which are essentially a representation of computation steps as functions).

By the way, we don't bother here to divide our grammars and semantic equations into submodules, because of the small scale of the example language. See [11] for a medium-sized example (a substantial sublanguage of ADA) where the use of submodules is advantageous.

## 6.1 Expressions

### 6.1.1 /Example/Abstract Syntax

**grammar:**

(1) Expression = Literal | ⟦ Monadic-Operator Expression ⟧ |

⟦ Expression Dyadic-Operator Expression ⟧ |
⟦ "if" Expression "then" Expression "else" Expression" ⟧ |
□ .

(2) Literal = "true" | "false" | Numeric-Literal |
Character-Literal | String-Literal .

**needs: Data Notation/Characters/Alphanumerics** ( digit ).

(3) Numeric-Literal = ⟦ digit$^+$ ⟧ .

(4) Character-Literal = □ .

(5) String-Literal = □ .

(6) Monadic-Operator = "¬" | "−" .

(7) Dyadic-Operator = "∧" | "∨" | "+" | "−" | "∗" | "=" .

The occurrences of □ above allow further constructs to be inserted in their place later. We could get the same flexibility by using inclusions instead of equations—then omitting the □s. Actually, we shall not bother to specify the details of Character-Literal and String-Literal at all.

By the way, a / at the beginning of a module title prevents the module from inheriting the titles of higher-level sections, i.e., the title is 'absolute' rather than 'relative'.

### 6.1.2 /Example/Semantic Functions

**needs: Abstract Syntax, Semantic Entities** .

**introduces:** evaluate _ , the value of _ ,
the monadic-operation-result of _ , the dyadic-operation-result of _ .

• evaluate _ :: Expression →
[give a value] act & [perhaps using nothing] act .

The above functionality assertion is not formally necessary. It is actually a consequence of the semantic equations below. But it does provide useful documentation about the sort of semantic entity that expressions denote. In particular, it confirms that expression evaluation cannot diverge or affect storage.

(1) evaluate $L$:Literal = give the value of $L$ .

(2) evaluate ⟦ $O$:Monadic-Operator $E$:Expression ⟧ =
evaluate $E$ then give the monadic-operation-result of $O$ .

(3) evaluate ⟦ $E_1$:Expression $O$:Dyadic-Operator $E_2$:Expression ⟧ =
| evaluate $E_1$ and evaluate $E_2$
then give the dyadic-operation-result of $O$ .

The use of and indicates that the order of expression evaluation is implementation-dependent. In the absence of side-effects (and abnormal termination) all orders lead to the same result. Sometimes languages allow side-effects and insist on left-to-right order of evaluation, which could be specified by using and then instead of and above.

(4) evaluate ⟦ "if" $E$:Expression "then" $E_1$:Expression "else" $E_2$:Expression ⟧ =
    evaluate $E$ then
    | | check (it is true) then evaluate $E_1$
    | or
    | | check (it is false) then evaluate $E_2$ .

The term '(it is true)' could be replaced by 'the given truth-value', or by 'there is given true'. Notice that the enclosing action fails unless evaluate $E$ gives a truth-value. Thus some *type-checking* is implicit in the semantic equation. Full type-checking could be specified separately in a *static* action semantics, such that the actions given by the *dynamic* semantics specified here would be infallible for statically-correct programs.

check $D$ doesn't give any data, and evaluate $E$ doesn't refer to given data, so it doesn't matter whether we combine them using then, or and then.

- the value of _ :: Literal → value .

(5) the value of "true" = true .

(6) the value of "false" = false .

(7) the value of ⟦ $d$:digit$^+$ ⟧ = bounded decimal (string of $d$) .

Here, we take a short-cut, using the standard operation decimal _ :: string → natural, which is provided by Data Notation, rather than introducing a corresponding semantic function. bounded _ is specified in **Semantic Entities** below. See [11] for a loose specification of approximate real arithmetic.

- the value of _ :: Character-Literal → character .
- the value of _ :: String-Literal → string .

Both the syntax and semantics of character and string literals are left open.

- the monadic-operation-result of _ :: Monadic-Operator →
    [perhaps using a given operand] dependent result .

(8) the monadic-operation-result of "¬" = not the given truth-value .

(9) the monadic-operation-result of "−" = the bounded negation of the given number .

bounded is actually redundant here, as bounds on numbers are specified to be symmetric.

- the dyadic-operation-result of _ :: Dyadic-Operator →
    [perhaps using a given (operand, operand)] dependent result .

(10) the dyadic-operation-result of "∧" =
    both of (the given truth-value#1, the given truth-value#2) .

(11) the dyadic-operation-result of "∨" =
    either of (the given truth-value#1, the given truth-value#2) .

Recall that the _ and of _ are identity on data.

Below, the application of bounded ensures that nothing is yielded when the result would be out of bounds, i.e., not of sort number.

(12) the dyadic-operation-result of "+" =
the bounded sum of (the given number#1, the given number#2) .

(13) the dyadic-operation-result of "−" =
the bounded difference of (the given number#1, the given number#2) .

(14) the dyadic-operation-result of "∗" =
the bounded product of (the given number#1, the given number#2) .

(15) the dyadic-operation-result of "=" =
the given operand#1 is the given operand#2 .

### 6.1.3 /Example/Semantic Entities

**includes: Action Notation [11, Appendix B]** .

Action Notation includes general Data Notation (without any commitment to a particular character set).

#### 6.1.3.1 Sorts

**needs: Values** .

(1) datum = value | □ .

#### 6.1.3.2 Values

**needs: Numbers** .

(1) value = truth-value | number | character | string | □ .

(2) operand = truth-value | number | character .

(3) result = truth-value | number .

The order and grouping of sort unions is immaterial. Contrast this flexibility with the rigidity of domain sums in Denotational Semantics [10]!

The differences between *characteristic* sorts such as value, operand, and result reveal quite a lot about the described language [16].

#### 6.1.3.3 Numbers

**introduces:** number , bounded _ , ordinal _ , bound .

**includes: Data Notation/Instant/Distinction** ( number *for* s , _ is _ ) .

This translated specification requires _ is _ to be a partial equality operation on number.

(1) bounded _ :: natural → number (*partial*) .
(2) ordinal _ :: number → natural (*partial*) .
(3) bound : natural .
(4) _ is _ :: number, number → truth-value (*total*) .
(5) number = [min negation of bound] [max bound] integer .
(6) bounded $n$:natural = $n$ & a number .
(7) ordinal $n$:number = $n$ & a natural .
**open** .

The above module must not be closed, because bound has been left as an unspecified natural number.

We are exploiting the standard arithmetical operations provided by Data Notation, and using sort intersection to bound the results. Some programming languages require several disjoint types of numbers, with various coercions between them. This takes somewhat longer to specify, as illustrated in [11].

## 6.2 Declarations

### 6.2.1 /Example/Abstract Syntax *(continued)*

**grammar:**

(1) Constant-Declaration = ⟦ "val" Identifier "=" Expression ⟧ .

(2) Constant-Declarations = ⟨ Constant-Declaration ⟨ ";" Constant-Declaration ⟩* ⟩ .

Thus *CD*:Constant-Declarations is a tuple of Constant-Declaration trees, separated by ";" components.

(3) Expression = □ | Identifier |
⟦ "let" Constant-Declarations "in" Expression ⟧ |
⟦ "fun" "(" Parameter-Declaration ")" Expression ⟧ |
⟦ Expression "(" Expression ")" ⟧ .

Various programming languages allow functions to be *declared*, i.e., bound to identifiers. Often—but not in ADA, for example—functions may also be passed as *arguments* to other functions. But only in a few languages (such as STANDARD ML) is it possible to *express* functions directly, as here, without necessarily binding them to identifiers.

(4) Parameter-Declaration = ⟦ "val" Identifier ⟧ | □ .

Functions in programs resemble mathematical functions: they return values when applied to arguments. In programs, however, the evaluation of arguments may diverge, so it is necessary to take into account not only the relation between argument values and result values, but also the stage at which an argument expression is evaluated: straight away, or when (if) ever the value of the argument is required for calculating the result of the application. This is generally indicated by the declaration of the parameter, rather than by the call. For instance, "val" in a parameter declaration above is supposed to indicate immediate evaluation, i.e., call-by-value. Further forms of parameter declaration are introduced in connection with procedures, in Section 6.4.

**introduces:** Identifier .

**needs:** **Data Notation/Characters/Alphanumerics** ( letter , digit ) .

(5) Word = ⟦ letter (letter | digit)* ⟧ .

(6) Word = Reserved-Word | Identifier (*disjoint*) .

(7) Reserved-Word = "if" | "then" | "else" | "true" | "false" |
"let" | "in" | "fun" | "val" | □ .

The use of the attribute *disjoint* above essentially specifies Identifier to be the difference between Word and Reserved-Word. (Sort difference is not monotonic, so it cannot be specified as an operation in our meta-notation.)

### 6.2.2 /Example/Semantic Functions *(continued)*

**introduces:** declare _ , pass _ , the token of _ .

- declare _ :: Constant-Declarations →
  [bind] act & [perhaps using current bindings] act .

(1) declare ⟦ "val" $I$:Identifier "=" $E$:Expression ⟧ =
  evaluate $E$ then bind the token of $I$ to the given value .

$E$ is *not* in the scope of the binding produced for $I$. The specification of self- and mutually-referential bindings is not illustrated here. It involves hybrid information processing, as explained in [11].

(2) declare ⟨ $CD_1$:Constant-Declaration ";" $CD_2$:Constant-Declarations ⟩ =
  declare $CD_1$ before declare $CD_2$ .

The use of before allows $CD_2$ to refer to bindings produced by $CD_1$. We would specify so-called simultaneous declarations by using and instead.

- evaluate _ :: Expression →
  [give a value or diverge] act & [perhaps using current bindings] act .

Note that we have had to change our assertion about the sort of expression denotations, to accommodate the new constructs. But we don't need to change any of our previously-specified semantic equations for expressions.

(3) evaluate $I$:Identifier = give the value bound to the token of $I$ .

(4) evaluate ⟦ "let" $CD$:Constant-Declarations "in" $E$:Expression ⟧ =
  furthermore declare $CD$ hence evaluate $E$ .

furthermore allows $E$ to refer to nonlocal bindings that are not overridden by $CD$. Remember that prefix combinators have higher precedence than infix ones.

(5) evaluate ⟦ "fun" "(" $PD$:Parameter-Declaration ")" $E$:Expression ⟧ =
  give a function of the closure of an abstraction of
  | furthermore pass $PD$ hence
  | | evaluate $E$ then give the result yielded by it .

The use of closure here ensures static bindings for the identifiers in $E$. By moving it to the following equation, we would specify dynamic scopes. By leaving it out altogether, we would specify that references to identifiers are local to functions.

(6) evaluate ⟦ $E_1$:Expression "(" $E_2$:Expression ")" ⟧ =
  | evaluate $E_1$ and evaluate $E_2$
  then enact the application of the function-abstraction of the given function#1
    to the given argument#2 .

We see that the parameter of the function is evaluated before passing it to the enaction of the function-abstraction. This is often referred to as 'call-by-value', at least in connection with functional programming languages. An alternative would be to delay evaluation until the parameter is *used*, which is referred to as call-by-name. The main difference it makes to the semantics of expressions is that an evaluation which doesn't terminate here *might* then terminate; the values given on termination are the same.

Only a few programming languages (e.g., ALGOL60) provide call-by-name parameters. Much the same effect, however, can be achieved by passing a (parameterless) function as a parameter, and applying it (to no parameters) wherever its value is required. In fact that corresponds closely to how we would specify call-by-name in an action semantic description: pass an abstraction incorporating the parameter evaluation, and enact it whenever the formal parameter is evaluated. You might like to work out the details for yourself. (Omit call-by-value for now. We illustrate a technique for dealing with several parameter passing modes simultaneously later, in Section 6.4.)

Another possible mode of parameter passing is to memorize the value of the parameter the first time it is evaluated, if ever. This is referred to as call-by-need. There is no implementation-independent semantic difference between call-by-need and call-by-name, although careful use of Action Notation can make the *intended* implementation technique apparent.

- pass _ :: Parameter-Declaration →
  [bind] act & [perhaps using a given argument | current bindings] act .

(7) pass ⟦ "val" *I*:Identifier ⟧ = bind the token of *I* to the given argument .

- the token of _ :: Identifier → token .

(8) the token of *I*:Identifier = *I* .

A common alternative semantics for identifiers is to ignore case differences between them, by mapping all letters in *I* to (say) upper-case.

### 6.2.3 /Example/Semantic Entities *(continued)*

#### 6.2.3.1 Sorts *(continued)*

(1) bindable = value | □ .

(2) token = string of (letter, (letter | digit)*) .

#### 6.2.3.2 Values *(continued)*

**needs: Functions** .

(1) value = □ | function | □ .

#### 6.2.3.3 Functions

**needs:** **Arguments** .

**introduces:** function , function of _ , function-abstraction _ .

(1) function of _ :: abstraction → function (*partial*) .

(2) function-abstraction _ :: function → abstraction (*total*) .

(3) $A$ : [give a result and perhaps diverge] act ;
$A$ : [perhaps using a given argument | current storage] act ⇒
a function of an abstraction of $A$ : function .

(4) $f$ = a function of $a$:abstraction ⇒
the function-abstraction of $f$:function = $a$ .

The primary effect of function of _ is to make a *tagged* copy of an abstraction, where the tag ensures that functions do not overlap with other sorts of data (such as procedures). Together with function-abstraction _ it provides an interface that hides the representation of functions from other modules.

#### 6.2.3.4 Arguments

**needs:** **Values** .

(1) argument = value | □ .

The characteristic sort argument is extended in Section 6.4. Its relation to the sort bindable indicates how closely parameter declarations might correspond to ordinary declarations.

The next section extends our example language with familiar statements and variable declarations.

## 6.3 Statements

#### 6.3.1 /Example/Abstract Syntax *(continued)*

**grammar:**

(1) Statement = "skip" | ⟦ Name ":=" Expression ⟧ |
⟦ "if" Expression "then" Statement ⟧ |
⟦ "while" Expression "do" Statement ⟧ | ⟦ "stop" ⟧ |
⟦ "begin" ⟨ Variable-Declarations ";" ⟩? Statements "end" ⟧ |
□ .

(2) Statements = ⟨ Statement ⟨ ";" Statement ⟩* ⟩ .

(3) Variable-Declaration = ⟦ "var" Identifier ":" Type ⟧ .

(4) Variable-Declarations = ⟨ Variable-Declaration ⟨ "," Variable-Declaration ⟩* ⟩ .

(5) Type = "bool" | "num" | ⟦ Type "[" Numeric-Literal "]" ⟧ .

(6) Name = Identifier | ⟦ Name "[" Expression "]" ⟧ .

(7) Expression = □ | ⟦ Expression "[" Expression "]" ⟧ .

The types allow nested arrays, for instance "num[10][20]". Similarly, expressions allow iterated indexing, for instance "a[3][7]". Notice that Name is a subsort of Expression.

### 6.3.2 /Example/Semantic Functions *(continued)*

**introduces:** execute _ , establish _ , relinquish _ , the type of _ , access _ .

- execute _ :: Statements →
  [complete or escape or diverge] act &
  [perhaps using current bindings | current storage] act .

(1) execute "skip" = complete .

(2) execute ⟦ $N$:Name ":=" $E$:Expression ⟧ =
  | access $N$ and evaluate $E$
  then assign the given value#2 to the given variable#1 .

assign _ to _ is specified in **Semantic Entities/Variables** below.

(3) execute ⟦ "if" $E$:Expression "then" $S$:Statement ⟧ =
  evaluate $E$ then
  | | check (it is true) then execute $S$
  | or check (it is false) .

(4) execute ⟦ "while" $E$:Expression "do" $S$:Statement ⟧ =
  unfolding
  | evaluate $E$ then
  | | | check (it is true) then execute $S$ then unfold
  | | or check (it is false) .

When $S$ is "stop" above, the escape causes the remaining unfolding to be ignored—and, in the absence of any traps, terminates the entire program. Different kinds of escape can be distinguished by giving them some data.

(5) execute "stop" = escape .

(6) execute ⟦ "begin" $S$:Statements "end" ⟧ = execute $S$ .

(7) execute ⟦ "begin" $VD$:Variable-Declarations ";" $S$:Statements "end" ⟧ =
  furthermore establish $VD$ hence
  | execute $S$ and then relinquish $VD$ .

The relinquishing of local variables only affects semantics when the supply of cells is finite. But it doesn't hurt to specify it, when the intention is for local variables to be reusable.

(8) execute ⟨ $S_1$:Statement ";" $S_2$:Statements ⟩ = execute $S_1$ and then execute $S_2$ .

- establish _ :: Variable-Declarations →
  [bind and store] act & [perhaps using current storage] act .

(9) establish ⟦ "var" $I$:Identifier ":" $T$:Type ⟧ =
  allocate a variable for the type of $T$ then
  bind the token of $I$ to it .

allocate _ for _ is specified in **Semantic Entities/Variables** below.

(10) establish ⟨ $VD_1$:Variable-Declaration "," $VD_2$:Variable-Declarations ⟩ =
establish $VD_1$ and establish $VD_2$ .

- relinquish _ :: Variable-Declarations →
[store] act & [perhaps using current bindings | current storage] act .

(11) relinquish ⟦ "var" $I$:Identifier ":" $T$:Type ⟧ =
dispose of the variable bound to the token of $I$ .

(12) relinquish ⟨ $VD_1$:Variable-Declaration "," $VD_2$:Variable-Declarations ⟩ =
relinquish $VD_1$ and relinquish $VD_2$ .

- the type of _ :: Type → type .

(13) the type of $T$:("bool" | "num") = simple-type .

(14) the type of ⟦ $T$:Type "[" $N$:Numeric-Literal "]" ⟧ =
array-type of (the type of $T$, the value of $N$) .

If we were to let types be declared, we would have to change their semantics from data to dependent data.

- access _ :: Name →
[give a variable or diverge] act &
[perhaps using current bindings | current storage] act .

(15) access $I$:Identifier = give the variable bound to the token of $I$ .

(16) access ⟦ $N$:Name "[" $E$:Expression "]" ⟧ =
| access $N$ and evaluate $E$
then give the component (the given array-variable#1, the given index-value#2) .

- evaluate _ :: Expression →
[give a value or diverge] act &
[perhaps using current bindings | current storage] act .

(17) evaluate $I$:Identifier =
give the value bound to the token of $I$ or
give the value assigned to the variable bound to $I$ .

The above equation *replaces* a previously-specified one! The change, though, is merely adding an alternative action, which is almost as easy as adding a fresh semantic equation. What is really remarkable is that *no other changes to the semantic equations were needed at all* when adding variables and statements to a functional language. The contrast between Action Semantics and conventional Denotational Semantics in this respect could hardly be more vivid.

(18) evaluate ⟦ $E_1$:Expression "[" $E_2$:Expression "]" ⟧ =
| evaluate $E_1$ and evaluate $E_2$
then give the component (the given array-value#1, the given index-value#2) .

We expect evaluate $N$:Name to be equivalent to access $N$ then give the value assigned to the given variable. We do not specify this as a semantic equation, though, because it would overlap with the above equation, and thereby cast doubt on the well-definedness of the semantic functions.

### 6.3.3 /Example/Semantic Entities *(continued)*

#### 6.3.3.1 Sorts *(continued)*

**needs:** **Variables** .

(1) datum = □ | variable .

(2) storable = truth-value | number .

#### 6.3.3.2 Values *(continued)*

**needs:** **Arrays** .

(1) value = □ | array-value | □ .

#### 6.3.3.3 Variables

**needs:** **Arrays, Types** .

**introduces:** variable , assign _ to _ , the _ assigned to _ , allocate _ for _ , dispose of _ .

(1) variable = cell | array-variable .

(2) assign _ to _ :: dependent value, dependent variable →
[perhaps store] act .

(3) the _ assigned to _ :: value, dependent variable →
[perhaps using current storage] dependent value .

(4) allocate _ for _ :: variable, dependent type →
[give a variable and perhaps store] act .

(5) dispose of _ :: dependent variable → [perhaps store] act .

(6) assign ($d_1$:dependent value) to ($d_2$:dependent variable) =
store the storable yielded by $d_1$ in the cell yielded by $d_2$ or
assign the array-value yielded by $d_1$ to the array-variable yielded by $d_2$ .

(7) the ($v \leq$value) assigned to ($d$:dependent variable) =
the ($v$ & storable) stored in the cell yielded by $d$ |
the ($v$ & array-value) assigned to the array-variable yielded by $d$ .

(8) allocate ($v \leq$variable) for ($d$:dependent type) =
check there is the simple-type yielded by $d$ and then
allocate a cell or
allocate a ($v$ & array-variable) for the array-type yielded by $d$ .

(9) dispose of ($d$:dependent variable) =
unreserve the cell yielded by $d$ or
dispose of the array-variable yielded by $d$ .

#### 6.3.3.4 Types

**needs:** **Arrays** .

(1) type = simple-type | array-type .

(2) simple-type : type .

**closed** .

#### 6.3.3.5 Arrays

**introduces:** array , index-value , array-value , array-variable ,
array of _ , component _ , array-type , array-type of _ ,
upper index-value _ , component-type _ .

**needs:** **Values , Variables , Types , Numbers** .

(1) array = array-value | array-variable .

(2) index-value ≤ number .

(3) array of _ :: value* → array-value ($total$) ,
variable* → array-variable ($total$) .

(4) component _ :: (array-value, index-value) → value ($partial$) ,
(array-variable, index-value) → variable ($partial$) .

(5) array-type of _ :: (type, index-value) → array-type ($total$) .

(6) upper index-value _ :: array-type → index-value ($total$) .

(7) component-type _ :: array-type → type ($total$) .

(8) $a$ = array of $v$ ⇒
component ($a$:array, $i$:index-value) = component#(ordinal $i$) of $v$ .

(9) $t$ = array-type of ($t'$, $n$) ⇒
component-type $t$:array-type = $t'$ ; upper index-value $t$:array-type = $n$ .

You should skip the details below on a first reading, as they are a bit tedious. See [11] for arrays with offsets (i.e., lower index bounds other than 1) and for a similar treatment of record variables. Such specifications can easily be reused in the semantic descriptions of other languages that involve structured variables, although minor variations in the operations may be required.

**privately introduces:** components _ , respectively assign _ to _ ,
the values respectively assigned to _ ,
allocate component-variables for _ up to _ , dispose of all _ .

(10) assign ($d_1$:dependent array-value) to ($d_2$:dependent array-variable) =
respectively assign the components of $d_1$ to the components of $d_2$ .

(11) the ($v$≤array-value) assigned to ($d$:dependent array-variable) =
the $v$ yielded by array of
the values respectively assigned to the components of $d$ .

(12) allocate ($v$≤array-variable) for ($d$:dependent array-type) =
allocate component-variables for the component-type of $d$
to the ordinal of the upper index-value of $d$
then give array of the given variable(s)? .

(13) dispose of ($d$:dependent array-variable) = dispose of all the components of $d$ .

(14) $a$ = array of $v$ ⇒ components of $a$:array = $v$ .

(15) respectively assign ($d_1$:dependent value*) to ($d_2$:dependent variable*) =
    | check ($d_1$ is ( )) and check ($d_2$ is ( ))
    or
    | assign the first of $d_1$ to the first of $d_2$ and
    | respectively assign the rest of $d_1$ to the rest of $d_2$ .

(16) the values respectively assigned to ($d$:dependent variable*) =
    when $d$ is ( ) then ( ) |
    (the value assigned to the first of $d$,
    the values respectively assigned to the rest of $d$) .

(17) allocate component-variables for ($d_1$:dependent type) to ($d_2$:dependent natural) =
    | check ($d_2$ is 0) and then give ( )
    or
    | check not ($d_2$ is 0) and then
    | | allocate component-variables for $d_1$ to the predecessor of $d_2$
    | | and allocate a variable for $d_1$ .

(18) dispose of all ($d$:dependent variable*) =
    check ($d$ is ( )) or
    | dispose of the first of $d$ and
    | dispose of all the rest of $d$ .

Let us conclude our example by adding procedure abstractions, with various modes of parameter-passing. The technique used below is of general use.

## 6.4 Procedures

### 6.4.1 /Example/Abstract Syntax *(continued)*

**grammar:**

(1) Expression = □ | ⟦ "proc" "(" Parameter-Declaration ")" Statement ⟧ .

(2) Parameter-Declaration = □ | ⟦ "var"? Identifier ":" Type ⟧ .

(3) Statement = □ | ⟦ Expression "(" Expression ")" ⟧ .

Procedure abstractions are much like function abstractions. The only differences are that the body of the abstraction is now a statement, rather than an expression, and that there are some new modes of parameter passing.

By the way, many programming languages do not allow a function to be expressed (or declared) directly: a procedure must be used instead, and the body of the procedure includes a special statement that determines the value to be returned. In ALGOL60 and PASCAL, this statement looks like an assignment to the function identifier! When such a function is applied in an expression, *side-effects* may occur: the evaluation of the expression changes storage, as well as giving a value.

As with functions, we only consider procedures with a single parameter, for simplicity. The new modes of parameter passing are call-by-reference, indicated by

the symbol "var", and call-by-copy (usually referred to, somewhat ambiguously, as call-by-value), indicated by the absence of "var".

The execution of a procedure body may have an effect on the state, by assignment to a nonlocal variable. With call-by-reference, moreover, assignment to the formal parameter changes the value of the nonlocal actual parameter variable (its *alias*) whereas with call-by-copy such an assignment merely modifies a local variable.

### 6.4.2 /Example/Semantic Functions *(continued)*

**introduces:** moderate _ , the mode of _ , relinquish _ .

(1) evaluate ⟦ "proc" "(" $PD$:Parameter-Declaration ")" $S$:Statement ⟧ =
    give a procedure of (the mode of $PD$,
        the closure of an abstraction of
        | furthermore pass $PD$ hence
        | | execute $S$ and then relinquish $PD$ ) .

(2) execute ⟦ $E_1$:Expression "(" $E_2$:Expression ")" ⟧ =
    evaluate $E_1$ then
    | give the procedure-abstraction of the given procedure and
    | | give the passing-mode of the given procedure then
    | | moderate $E_2$ then give the argument yielded by it
    then enact the application of the given procedure-abstraction#1
                                to the given argument#2 .

Compare the above action with the one previously given for function application. The action moderate $E_2$ is defined below to either access or evaluate $E_2$, depending on the given mode. Clearly we now have to evaluate $E_1$ before $E_2$, in order to obtain the parameter passing mode.

- pass _ :: Parameter-Declaration →
    [bind and perhaps store] act & [perhaps using a given argument] act .

(3) pass ⟦ "var" $I$:Identifier ":" $T$:Type ⟧ =
    bind the token of $I$ to the given variable .

(4) pass ⟦ $I$:Identifier ":" $T$:Type ⟧ =
    | give the given value and
    | allocate a variable for the type of $T$
    then
    | bind the token of $I$ to the given variable#2 and
    | assign the given value#1 to the given variable#2 .

(5) pass ⟦ "val" $I$:Identifier ⟧ =
    bind the token of $I$ to the given value .

- relinquish _ :: Parameter-Declaration →
    [perhaps store] act & [perhaps using current bindings] act .

(6) relinquish ⟦ "var" $I$:Identifier ":" $T$:Type ⟧ = complete .

(7) relinquish ⟦ $I$:Identifier ":" $T$:Type ⟧ =
    dispose of the variable bound to the token of $I$ .

(8) relinquish ⟦ "val" $I$:Identifier ⟧ = complete .

- the mode of _ :: Parameter-Declaration → mode .

(9) the mode of ⟦ "var" $I$:Identifier ":" $T$:Type ⟧ = reference-mode .

(10) the mode of ⟦ $I$:Identifier ":" $T$:Type ⟧ = copy-mode .

(11) the mode of ⟦ "val" $I$:Identifier ⟧ = constant-mode .

- moderate _ :: Expression →
  [give an argument or diverge] act &
  [perhaps using a given mode | current bindings | current storage] act .

(12) moderate $E$:Expression =
  | check (the given mode is reference-mode) then access $E$
  or
  | check either (it is copy-mode, it is constant-mode) then evaluate $E$ .

What should happen if a procedure whose parameter is to be passed by reference gets called with an actual parameter expression other than a name? A static semantics for our example language would presumably reject programs containing such constructs. But the restriction is not context-free, so we cannot represent it in our abstract syntax (without abandoning a simple relation to context-free concrete syntax, that is). Instead, we extend access _ from Name to Expression, as follows.

- access _ :: Expression →
  [give a variable or diverge] act &
  [perhaps using current bindings | current storage] act .

(13) $E$ & Name = nothing ⇒ access $E$ = fail .

The semantic entities specified below are a simple generalization of those used before to represent function abstractions.

### 6.4.3 /Example/Semantic Entities *(continued)*

#### 6.4.3.1 Values *(continued)*

**needs:** **Procedures** .

(1) value = □ | procedure .

#### 6.4.3.2 Procedures

**needs:** **Arguments** .

**introduces:** procedure , procedure of _ , passing-mode _ , procedure-abstraction _ .

(1) procedure of _ :: (mode, abstraction) → procedure (*partial*) .

(2) procedure-abstraction _ :: procedure → abstraction (*total*) .

(3) passing-mode _ :: procedure → mode (*total*) .

(4) $m$ : mode ;
  $A$ : [perhaps store and perhaps escape and perhaps diverge] act ;
  $A$ : [perhaps using a given argument | current storage] act ⇒
  procedure of ($m$, an abstraction of $A$) : procedure .

(5) $p$ = procedure of ($m$:mode, $a$:abstraction) ⇒
the passing-mode of $p$:procedure = $m$ ;
the procedure-abstraction of $p$:procedure = $a$ .

#### 6.4.3.3 Arguments *(continued)*

**needs: Variables** .

(1) argument = □ | variable .

(2) mode = reference-mode | copy-mode | constant-mode (*individual*) .

(3) disjoint (reference-mode, copy-mode, constant-mode) = true .

So much for the examples.

# 7 Conclusion

This introduction to Action Semantics explained meta-notation, Data Notation, and Action Notation. It also gave some basic examples of action semantic descriptions. Let us conclude with a critical assessment of the success of the approach.

First of all, there is the question of its *universality*: can Action Semantics cope—easily—with all kinds of programming language? Previous experiments (in various versions of Action Notation) indicate that there is no problem with conventional, PASCAL-like languages,[2] nor with eager functional languages like STANDARD ML. At present, though, we have no nice way of describing languages with explicit manipulation of continuations, such as SCHEME. Lazy functional languages have not been tried yet, but should yield to the technique sketched for call-by-need in the previous section. Languages with synchronous communication, such as OCCAM2 and CCS, require asynchronous action protocols that reveal how synchronicity between distributed processes can be achieved—in fact it can't, in general, unless one introduces some extraneous arbiters, or breaks symmetry some other way! The sharing of storage between distributed agents is not supported directly by Action Notation, but can be represented by separate processes—as can communication channels. There is no support for real-time either. Object-oriented languages tend to have rather complicated scope rules, but otherwise pose no special problems, it seems. How about logic programming languages? Action Semantics is for explicating the intended *operational* understanding of a language; preliminary studies indicate that it is possible to give an action semantic description of the usual *procedural* semantics of PROLOG (leaving out "assert" and "retract", which are inherently non-compositional). But it doesn't support a description of the *declarative* semantics of logic programming languages.

Anyway, if major inadequacies of Action Notation are discovered, it may be possible to embellish actions with new facets of information processing without disturbing the old ones, and without undermining the overall approach.

Another aspect of the success of Action Semantics is the question of its *acceptability*, both to theoretical computer scientists and to practical programmers. From

[2] A showcase action semantics for ISO Standard Pascal is currently being prepared for publication.

a theoretical point of view, the foundations of Action Notation have indeed been established—by an operational semantics and derived equivalences—but the theory of actions is still quite underdeveloped and intractable, especially compared to the theory of continuous functions that has been provided for Denotational Semantics. All contributions to developing the theory of actions are most welcome!

It remains to be seen whether practical programmers (such as language designers, implementors, and standardizers) will abandon informal descriptions of programming languages, and if so, whether action semantic descriptions will win their favour. Surely the unique extensibility, modifiability, and readability of action semantic descriptions—obtained without sacrificing formality or compositionality—indicate that this approach does provide a viable alternative to informal descriptions. By the way, action semantic descriptions have already been found useful in studies of semantics-based compiler generation. Tools for the input, browsing, checking, and interpretation of descriptions are currently being developed.

**Acknowledgments:** David Watt has been collaborating on the development of Action Semantics since 1985, especially regarding experimentation with the details of Action Notation and its use. Paddy Krishnan, Jens Palsberg, and Bernhard Steffen have provided useful suggestions about how best to present Action Semantics. The reactions of the Summer School participants to the preliminary version of these notes, and to my lectures, indicated several places where further clarification was necessary. The development of action semantics was partially funded by the Danish Science Research Council DART project (5.21.08.03).

# References

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

[2] J. A. Goguen and J. Meseguer. Order-sorted algebra: Algebraic theory of polymorphism. *Journal of Symbolic Logic*, 51:844–845, 1986. Abstract.

[3] 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.

[4] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Lab., SRI International, 1988.

[5] E. Moggi. Computational lambda-calculus and monads. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 14–23. IEEE, 1989.

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

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

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

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

[10] P. D. Mosses. Denotational semantics. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[11] P. D. Mosses. Action Semantics. Lecture Notes, Version 9 (a revised version is to be published by Cambridge University Press in the series *Tracts in Theoretical Computer Science*), 1991.

[12] P. D. Mosses. A practical introduction to denotational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 1–49. Springer-Verlag, 1991.

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

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

[15] C. Strachey. Fundamental concepts of programming languages. Lecture Notes for a NATO Summer School, Copenhagen. Available from Programming Research Group, University of Oxford, 1967.

[16] C. Strachey. The varieties of programming language. In *Proc. International Computing Symposium*, pages 222–233. Cini Foundation, Venice, 1972. A revised and slightly expanded version is Tech. Mono. PRG–10, Programming Research Group, University of Oxford, 1973.

# Appendix

The following comments provide answers to most of the questions posed about Data Notation in Section 3.

**Numbers/Naturals/Basics:**

The individual natural numbers 0, successor 0, ..., are all just above nothing, i.e., atoms of the sort lattice. This is because the attribute *total* includes both *strict* and *linear*, and the only way of expressing a value between an individual and nothing is by use of intersection & of individuals. For instance we have:

$$\begin{aligned}&\mathsf{nothing} \leq \\ &\quad \mathsf{1 \ \& \ 2} = \\ &\quad \mathsf{successor\ 0 \ \&\ successor\ 1} = \\ &\quad \mathsf{successor\ (0 \ \&\ 1)} \leq \\ &\quad \mathsf{successor\ (0 \ \&\ successor\ natural)} \leq \\ &\quad \mathsf{successor\ (0 \ \&\ positive\text{-}integer)} = \\ &\quad \mathsf{successor\ nothing} = \mathsf{nothing}\ .\end{aligned}$$

Hence 1 & 2 = nothing, and similarly for all other pairs of individual natural numbers. The only expressible vacuous sort is the bottom of the lattice, nothing.

We have only successor natural $\leq$ positive-integer; the reverse inclusion is *not* a consequence of the specification. Since these sorts have the same extension, they could be equated, if desired.

**Tuples/Basics:**

We have (0,1) : $(0 \mid 1)^*$, but not (0,1) : $(0^*) \mid (1^*)$. Hence the operation $\_^*$ *cannot* be *linear*.

**Numbers/Naturals/Specifics:**

Any term of the form sum (successor$^m$ 0, successor$^n$ 0) can be converted to the term successor$^{m+n}$ 0 by first using axiom (3) *from right to left* to eliminate successor in favour of sum, and then using the associativity of sum to regroup the summation, before finally using axiom (3) from left to right.

Any cofinite sort of natural numbers can be expressed in **Basics** by using a term of the form $s \mid \mathsf{successor}^n\mathsf{natural}$, where $s$ is a finite union of individual natural numbers. The sort of *even* natural numbers isn't directly expressible in **Basics**. It can be expressed in **Specifics** as product (2, natural).

# Contents