ISSN 0105-8517

Unified Algebras and Action Semantics

Peter D. Mosses

DAIMI PB – 272
December 1988

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 86 12 71 88 *Telex: 64767 aausci dk*

Abstract

The recently-developed framework of Unified Algebras is intended for axiomatic specification of abstract data types. In contrast, the somewhat older framework of Action Semantics (earlier known as "Abstract Semantic Algebras") is for denotational specification of programming languages. This paper gives an introduction to the main features of Unified Algebras and Action Semantics, and discusses the relation between them. The two frameworks both exploit nondeterminstic choice in unconventional ways.

A reformatted version of this report is to appear as an invited paper in the *Proceedings of STACS'89 (Paderborn, February 1989), Springer-Verlag Lecture Notes in Computer Science*; citations should refer to the Proceedings.

1 Introduction

The aim of this invited paper is to give an introduction to the author's work on two distinct, yet closely related, topics: "Unified Algebras", a recently-developed framework for the algebraic specification of abstract data types; and "Action Semantics", earlier known as "Abstract Semantic Algebras", a framework for the denotational specification of programming language semantics, which has been developed (partly in collaboration with David Watt, Glasgow) over the past decade. Unified algebras were originally developed to facilitate the specification of the semantic entities used in action semantics, although it seems that they may be of more general applicability. The notation used in action semantics is currently being revised to take full advantage of unified algebras. Both frameworks make essential use of a "join" operation, which corresponds closely to nondeterministic choice.

The framework of Unified Algebras was developed from the framework of "order-sorted algebras" [5,9,22], which underlies the OBJ specification language [4,6,11], and which was itself developed from "many-sorted algebras" [10,3].

With unified algebras there is a unified treatment of the "elements" of an abstract data type and their classifications into "sorts". In fact elements are treated as singleton sorts. Thus the operations of a unified algebra may take sorts and/or elements as arguments, and give sorts or elements as results. The immediate benefits of this generality are as follows:

- Ordinary operations on elements can be extended "element-wise" to sorts, so that for instance the successor operation maps the sort of natural numbers to the sort of positive integers.

- Partial operations can easily be accommodated: the vacuous sort represents the lack of a result, i.e., undefinedness.

- Operations that map elements to sorts correspond to "dependent" sorts, e.g., mapping a natural number n to the interval $[0 \mathinner{..} n]$, which is the sort of all natural numbers up to n.

- Operations that map sorts to sorts (not necessarily element-wise extensions of operations on elements) correspond to sort "constructors", for instance mapping two sorts to their union, or mapping a sort D to the sort of lists with components in D. Such operations allow a straightforward specification of polymorphism, unifying the notions of "parametric" and "inclusion" polymorphism: the sort of lists of D is a subsort of the sort of all lists.

Action Semantics was developed from Denotational Semantics [21, 24,23,12,19]. An action semantics for a programming language is a compositional mapping from abstract syntactic entities to abstract semantic entities called "actions". These actions have a more operational nature than the higher-order functions used as denotations in conventional Denotational Semantics: an action can be (notionally) "performed" so as to "process information". It is quite straightforward to represent the semantics of most programming constructs by actions; action semantics has other pragmatic virtues as well. However, the theory of actions is not as "powerful" as Scott's domain theory for higher-order functions.

The basis of Action Semantics is a "standard" notation for actions, called "Action Notation". It provides various primitive actions, such as computing an item of data from previously-computed data, checking that a predicate holds (otherwise "failing"), and storing data in a cell. Action Notation also provides a number of action combinators, including sequencing, interleaving, and—of special significance in relation to Unified Algebras—nondeterministic choice.

Of course, programming languages do not often have constructs whose semantics is "genuinely" nondeterministic, i.e., where an implementation should make some random choice each time the construct is executed. But they usually have some "implementation-dependent" features, for instance the order of evaluation of subexpressions. In Action Semantics, nondeterministic actions are used to represent such implementation-dependence, as well as genuine nondeterminism.

Action Notation enjoys various pleasant algebraic laws. While these laws were being specified (using a variant of OBJ) the following question arose: What is the essential difference between a sort of actions and a nondeterministic action? More generally, what is the difference between sort union and nondeterministic choice?

The answer seems to be that there is very little difference. Operations that map nondeterministic actions to nondeterministic actions correspond to operations from sorts to sorts. Increasing the nondeterminacy of an action cannot do anything but increase the nondeterminacy of any action in which it occurs, which corresponds to the operations on sorts preserving subsort inclusions.

This observation directly inspired the framework of unified algebras. Section 2 gives the details of unified algebras. Some results are stated; they are proved elsewhere [16]. Practical notation for basic specifications of (classes of) unified algebras is introduced—see [14] for further details, and for notation for modular specifications.

Section 3 presents Action Semantics. A substantial part of Action Notation is introduced formally, using the unified specification framework. The version of Action Notation given here differs in some details from previous versions [17,15], mainly due to taking advantage of new possibilities provided by the unified treatment of sorts and elements.

Throughout, the reader is assumed to be familiar with the general idea of algebraic specification of abstract data types. For Section 3, some familiarity with denotational semantics is useful. No familiarity with previous papers on Unified Algebras or Action Semantics is assumed.

2 Unified Algebras

To start with, let us recall the basic concepts of abstract data types, and relate them to unified algebras.

2.1 Concepts

A *data type* consists of a set of *elements* (such as numbers or lists) together with a collection of *operations* between elements—i.e., an algebra. An *abstract* data type is a *class* of algebras that share some properties.

In the so-called "algebraic" approach to specification of abstract data types, a basic specification consists of a *signature* and a set of logical *sentences*. The signature provides symbols for operations (constants are regarded as operations with no arguments). The satisfaction of the sentences provides properties of the operations. The specified class of algebras may consist of all algebras that have the named operations with the given properties, or it may be "constrained", e.g., to initial algebras.

When specifying an abstract data type algebraically, it is helpful to identify various *classifications* of elements, and to give for each operation, the relation between the classifications of its arguments and the classification of its result. If the classifications of the arguments of an operation are specialized, that of the result may also be specialized. In particular, when arguments are restricted to single element classifications, the result classification may be restricted to the result of applying the operation to these elements.

Classifications are usually treated as indices, called "sorts": the set of elements of an algebra is then a sort-indexed family of subsets, the operation symbols of the signature are indexed by the sorts of arguments and results. With a "many-sorted" algebra there is no intrinsic relation between the sorts: the subsets that they index may or may not overlap. With an "order-sorted" algebra, the signature defines a partial order on the set of sorts, which has to be respected by the inclusion relation between the subsets they index. In both the many-sorted and order-sorted frameworks, there is a sharp distinction between classifications and elements.

With unified algebras, however, classifications have the same status as elements—in particular, operations may be applied to classifications as well as to elements. Let us henceforth refer to classifications and elements together as *choices*, avoiding the words "type" and "sort", which have rather too many connotations already.

A unified algebra consists of a set of choices, with a distinguished subset of elements, together with constants that denote particular choices, and operations that map choices to choices. A unified algebra does not necessarily provide *all* possible choices between elements. However, the set of choices always includes the vacuous choice, Hobson's choices[1] of single elements, and all finite choices. The set of choices is always closed under (finitary) union and intersection.

Choices are partially ordered by *inclusion*: if c_1 and c_2 are choices, then '$c_1 \leq c_2$' asserts that c_1 is included in c_2. An important special case of inclusion is the *classification* relation: '$c_1 : c_2$' (which may be pronounced "c_1 is-a c_2") asserts that c_1 is the Hobson's choice of a single element, included in c_2. Different Hobson's choices are incomparable in the partial order. The vacuous choice, denoted '$\mathsf{nothing}$', is least in the partial order. The choice between two choices c_1, c_2, denoted '$c_1 \mid c_2$', is their least upper bound; their "agreement", denoted '$c_1 \& c_2$', is their greatest lower bound.

The set of choices between elements forms a distributive lattice [13]

[1]For the benefit of readers unfamiliar with this idiom: "Hobson's choice: option of taking the one offered or nothing [from T. Hobson, Cambridge carrier (d. 1631) who let out horses on this basis]." [2]

with a bottom. Note that the Hobson's choices need not be the so-called "atoms" of the lattice (i.e., "just above" the bottom); but choices between them and the bottom are not much use, as they cannot include any elements. More generally, choices need not be "extensional": two distinct choices may classify the same set of elements.

NB! Choice inclusion must not be confused with computational approximation in Scott domains [20]; in fact lattices here are *not* usually complete. Operations are monotonic, preserving choice inclusion, but not necessarily continuous.

The notation used in basic unified specifications is similar to that used in OBJ: the declarations of symbols for constants and operations may be combined with information about their relation to classification and inclusion. But no distinction is made between symbols that denote elements and those that denote (multiple) choices—except that we generally use "Capitalized" words for the latter. We exploit "mix-fix" notation to write the application of an operation symbol 'S_0_ ... _S_n' to terms '$T_1, \ldots, T_n$' as '$S_0 T_1 \ldots T_n S_n$'; e.g., we write 'if_then_else_(T,X,Y)' as 'if T then X else Y'. Variables, such as X, range over all choices; they do not need to be declared.

Let us briefly consider some examples, before proceeding to formal details. First, Figure 1 specifies the usual truth-values. The use of the ordinary arrow '$\rightarrow$' in the specification of 'not_' indicates that the operation is "total" (mapping elements to elements) and "strict" (mapping 'nothing' to itself). But 'if_then_else_' is non-strict in its second and third arguments (which need not be truth-values), so a different arrow '$\Rightarrow$' is used, merely indicating the inclusion relation between argument and result choices. Note that the long arrow '$\Longrightarrow$' in the last clause stands for implication. (Conjunction in clauses, illustrated below, is written ';'.)

Consider also the specification of natural numbers given in Figure 2. The equation for 'Natural' resembles a domain equation; but '_ | _' is associative, commutative, and idempotent, so it does not correspond exactly to the sum construction used in domain theory. The 'predecessor_' operation is partial on 'Natural': it may give 'nothing' when applied to an element, which is indicated by the arrow '$\rightsquigarrow$'. Notice that m and n are restricted to be elements in the clause involving 'sum' and 'product'; this avoids the second conclusion of the clause giving problems with multiple choices for m. The operation '[0 .. _]' provides intervals, which are included in each other in the obvious way.

constant	Truth-Value = true | false
constant	true : Truth-Value
constant	false : Truth-Value
operation	not_ : Truth-Value → Truth-Value true → false false → true
operation	if_then_else_ : Truth-Value, X, $Y \Rightarrow (X \mid Y)$ nothing, X, $Y \Rightarrow$ nothing defined elementary

if true then X else $Y = X$

if false then X else $Y = Y$

$(T\&\text{Truth-Value}) = \text{nothing} \Longrightarrow (\text{if } T \text{ then } X \text{ else } Y) = \text{nothing}$

Figure 1: Unified Specification of Truth Values

As a final example of unified algebraic specifications of familiar abstract data types, consider Figure 3. The specified properties of 'cons(_,_)' ensure that components of lists are always elements of 'Data'. Notice that the operation '_(of_)' provides classifications of lists according to classifications of components: 'l(of D)' is just the list l when all the components are included in D, otherwise it is 'nothing'. A more thorough specification of lists would use '_(of_)' to specify the polymorphic properties of the other operations. Incidentally, 'cons(Data,List)' classifies the non-nil lists.

2.2 Formalities

So much for the concepts underlying unified algebras. Let us now define *signatures*, *sentences*, *models*, and *satisfaction*, to obtain an appropriate "institution" [1,7,8] for unified algebras.

First, it is convenient to specialize the conventional notation for many-sorted algebras by eradicating sort-indexed sets, as follows.

Let Symbol be the set of *symbols* used to name constants and operations, partitioned into disjoint subsets Symbol_n, $n \geq 0$. Let Variable be a set of *variables*, disjoint from Symbol.

A *homogeneous algebraic signature* is simply a subset Σ of Symbol. We

```
constant    Natural = 0 | successor Natural
constant    0 : Natural
operation   successor_ : Natural → Natural
constant    Positive = successor Natural
operation   predecessor_ : Natural ⤳ Natural
                           Positive → Natural
                           0 ⤳ nothing
N ≤ Natural ⟹ predecessor(successor N) = N
operation   sum(_,_) : Natural² → Natural
                       Positive, Natural → Positive
                       associative commutative unit(0)
operation   product(_,_) : Natural² → Natural
                           Positive² → Positive
                           0, Natural → 0
                           associative commutative unit(successor 0)
| m : Natural ; n : Natural
⟹
| sum(m, successor n) = successor sum(m,n) ;
| product(m, successor n) = sum(m, product(m,n))
operation   [0 .. _] : Natural ⇒ Natural strict defined
n : Natural ⟹ [0 .. n] = n | [0 .. predecessor n]
```

Figure 2: Natural Numbers with Intervals

write Σ_n for $\Sigma \cap \mathsf{Symbol}_n$, for $n \geq 0$. A *homogeneous algebraic signature morphism* $\sigma : \Sigma \to \Sigma'$ is a family of maps $\sigma_n : \Sigma_n \to \Sigma'_n$ $(n \geq 0)$. We write $\sigma(f)$ for $\sigma_n(f)$, where $f \in \Sigma_n$.

A *homogeneous (Σ-)algebra* A consists of a set $|A|$ (of *choices*) and for each $f \in \Sigma_n$ a function $f_A : |A|^n \to |A|$ (called a constant when $n = 0$, otherwise an operation). A *(Σ-)homomorphism* $h : A \to B$ is a function from $|A|$ to $|B|$ such that for any $f \in \Sigma_n$ and $a_1, \cdots, a_n \in |A|$

$$h(f_A(a_1, \cdots, a_n)) = f_B(h(a_1), \cdots, h(a_n)).$$

So much for homogeneous algebras. Now for unified algebras.

$$
\begin{array}{ll}
\textsf{constant} & \textsf{Data} \\
\textsf{constant} & \textsf{List} = \textsf{nil} \ \mid \ \textsf{cons}(\textsf{Data}, \textsf{List}) \\
\textsf{constant} & \textsf{nil} : \textsf{List} \\
\textsf{operation} & \textsf{cons}(\_,\_) : \textsf{Data}, \textsf{List} \rightarrow \textsf{List} \\
\textsf{operation} & \textsf{head}\_ : \textsf{List} \leadsto \textsf{Data} \\
\textsf{operation} & \textsf{tail}\_ : \textsf{List} \leadsto \textsf{Data} \\
\multicolumn{2}{l}{\textsf{head nil} = \textsf{nothing} \ ; \ \textsf{tail nil} = \textsf{nothing}} \\
\multicolumn{2}{l}{\mid d : \textsf{Data} \ ; \ l : \textsf{List}} \\
\multicolumn{2}{l}{\Longrightarrow} \\
\multicolumn{2}{l}{\mid \textsf{head cons}(d, l) = d \ ; \ \textsf{tail cons}(d, l) = l} \\
\textsf{operation} & \_(\textsf{of}\ \_) : \textsf{List}, \textsf{Data} \Rightarrow \textsf{List} \\
\multicolumn{2}{l}{\mid D \leq \textsf{Data}} \\
\multicolumn{2}{l}{\Longrightarrow} \\
\multicolumn{2}{l}{\mid \textsf{nil}\ (\textsf{of}\ D) = \textsf{nil};} \\
\multicolumn{2}{l}{\mid \quad \mid d : \textsf{Data} \ ; l : \textsf{List}} \\
\multicolumn{2}{l}{\mid \quad \Longrightarrow} \\
\multicolumn{2}{l}{\mid \quad \mid \textsf{cons}(d, l)(\textsf{of}\ D) = \textsf{cons}(d \& D,\ l(\textsf{of}\ D))}
\end{array}
$$

Figure 3: Generic Lists

A *unified signature* is a homogeneous algebraic signature that includes the constant symbol 'nothing' and the binary operation symbols '$\_ \mid \_$' and '$\_\&\_$'. Unified signature morphisms are homogeneous signature morphisms that preserve the given symbols. We write **UniSign** for the set of unified signatures. Henceforth, let Σ always be a unified signature.

A *(Σ-)unified sentence* is a universal Horn clause with variables from Variable, operation symbols from Σ, and binary predicate symbols '$=$', '$\leq$', and '$:$'. We write **UniSen**(Σ) for the set of (Σ)-unified sentences. Unified signature morphisms extend to translations of unified sentences (leaving variables unchanged).

A *(Σ-)unified algebra A* is a homogeneous (Σ-)algebra A such that:

- $|A|$ is a *distributive lattice* with $\_ \mid \_{A}$ as join, $\_\&\_{A}$ as meet, and $\textsf{nothing}_A$ as bottom. Let $\leq_A$ denote the partial order of the lattice.
- There is a distinguished subset of incomparable values, $E_A \subseteq |A|$ (the *elements* of A). Note that E_A need not be the "atoms" of the

lattice.

- For each $f \in \Sigma$, the function f_A is monotone (in each argument) with respect to $\leq_A$.

A *(Σ-)unified homomorphism* is a (Σ-)homomorphism that respects the partial order and maps elements to elements. We write **UniAlg**(Σ) for the class of (Σ-)unified algebras.

The binary predicate symbols '=', '$\leq$', and ':' are interpreted as follows in a unified algebra A:

- $x = y$ holds iff x is identical to y;
- $x \leq y$ holds iff $x \leq_A y$;
- $x : y$ holds iff $x \in E_A$ and $x \leq_A y$.

An institution UNI of unified algebras can be defined in the usual way, in terms of the evident categories of unified signatures, unified sentences, unified algebras, and the standard notion of satisfaction for universal Horn clauses. In [16] it is shown that the institution is "liberal", and appropriate "data constraints" [8] are defined.

2.3 Specifications

We next define the syntax and semantics of *canonical* specifications, which correspond directly to unified signatures and sentences. Such specifications are adequate in theory, but somewhat tedious to use in practice. Therefore we proceed to extend the syntax with some convenient *abbreviations*, which allow us to write basic unified specifications that resemble the order-sorted signatures and sentences used in OBJ. (In [14] it is shown how this basic specification language can be extended to allow "modules" and "constraints", which are out of the scope of this paper.)

We don't bother to give an unambiguous concrete syntax for our specification language. Instead, we use ambiguous grammars to define its *abstract syntax*. The grammars are written in a minor variant of BNF: '$\geq$' stands for "produces", ' **|** ' stands for "alternatively", and terminal symbols are enclosed in quotation marks. (In [14] it is shown how such grammars can themselves be regarded as basic specifications, leading to a unified algebraic treatment of abstract syntax.)

Each non-terminal of a grammar generates a set of strings (of terminal symbols); the derivation trees for these strings—equipped with the tree construction operations—are (essentially) the desired abstract syntactic entities. For writing examples of specifications, we use parentheses and indentation to indicate which abstract syntactic entities are intended, when this is not clear from the context. (A thin vertical bar is used for emphasizing indentation—and hence grouping.)

The abstract syntax of canonical basic specifications is defined by the grammar given in Figure 4. The grammar does not define the micro-

$$
\begin{array}{|l}
p : \mathsf{Positive} \\
\Longrightarrow \\
\begin{array}{lcl}
\mathsf{Basic} & \geq & \text{“}\mathsf{constant}\text{”}\ \mathsf{Symbol}_0 \ \mid\ \text{“}\mathsf{operation}\text{”}\ \mathsf{Symbol}_p \ \mid \\
 & & \mathsf{Clause} \ \mid\ \mathsf{Basic}\ \mathsf{Basic} \qquad\qquad ; \\
\mathsf{Clause} & \geq & \mathsf{Formula} \ \mid\ \mathsf{Formula}\ \text{“}\Longrightarrow\text{”}\ \mathsf{Clause} \ ; \\
\mathsf{Formula} & \geq & \mathsf{Term}\ \mathsf{Relator}\ \mathsf{Term} \ ; \\
\mathsf{Relator} & \geq & \text{“}=\text{”} \ \mid\ \text{“}\leq\text{”} \ \mid\ \text{“}:\text{”} \ ; \\
\mathsf{Term} & \geq & \mathsf{Variable} \ \mid\ \mathsf{Symbol}_0 \ \mid\ \mathsf{Symbol}_p\ \mathsf{Terms}_p \ ; \\
\mathsf{Terms}_1 & \geq & \mathsf{Term} \ ; \\
\mathsf{Terms}_{p+1} & \geq & \mathsf{Term}\ \text{“},\text{”}\ \mathsf{Terms}_p
\end{array}
\end{array}
$$

Figure 4: Abstract Syntax of Canonical Specifications

syntax of symbols (‘Symbol_n’, $n \geq 0$) and variables (‘Variable’). For symbols, let us use strings of characters in this sans serif font, with the number of occurrences of the place-holder character ‘_’ determining the index (i.e., rank) of the symbol. For variables, let us use strings of letters in *this italic font*, optionally distinguished by numerical subscripts.

The grammar is not quite context-free: the indices on the nonterminal symbols ‘Symbol’ and ‘Terms’ ensure that operation symbols are only applied to the number of arguments indicated by their indices. Each ‘Symbol_n’ (for $n \geq 0$) and ‘Terms_p’ (for $p \geq 1$) may be regarded as a distinct nonterminal symbol, if desired.

A simple example of a canonical specification is given in Figure 5. (It corresponds roughly to the specification of truth-values given in Figure 1.) There is no need to disambiguate the grouping of the symbol declarations constant Truth-Value

constant true

constant false

true : Truth-Value

false : Truth-Value

Truth-Value = true | false

operation if_then_else_

$T \leq$ Truth-Value $\Longrightarrow$

(if T then X else Y) $\leq$ (X | Y)

if true then X else $Y = X$

if false then X else $Y = Y$

if nothing then X else Y = nothing

if (T | U) then X else Y =

(if T then X else Y) | (if U then X else Y)

T & Truth-Value = nothing $\Longrightarrow$

(if T then X else Y) = nothing

Figure 5: Canonical Specification of Truth Values

and the clauses, as it is semantically irrelevant (in fact juxtaposition of specifications is like choice: associative, commutative, and idempotent).

Now let us define the semantics of canonical specifications. First of all, a specification is said to be *complete* when all the constant and operation symbols occurring in terms (except for the reserved symbols 'nothing', '_ | _', and '_&_') are declared by 'constant S' or 'operation S'. We do not care to give a semantics for incomplete specifications, although it could be done.

Following Sannella and Tarlecki [18], the semantics of a complete specification B consists of two components: Sig$[\![B]\!]$, the signature specified by B; and $\mathsf{Alg}[\![B]\!]$, the class of algebras specified by B. We define:

$$\mathsf{Sig}[\![B]\!] = \{S \in \mathsf{Symbol} \mid S \text{ occurs in } B\} \cup \{\text{`nothing'}, \text{`\_ | \_'}, \text{`\_\&\_'}\}$$
$$\mathsf{Alg}[\![B]\!] = \{A \in \mathbf{UniAlg}(\mathsf{Sig}[\![B]\!]) \mid A \text{ satisfies all the clauses in } B\}.$$

It is left to "constraints" to restrict the class of unified algebras that satisfy a specification to "initial" (more generally, "freely-generated") algebras—see [16] for the details.

As may be seen from the specification of truth-values in Figure 5, canonical specifications are a bit tedious to use. Let us introduce some formal abbreviations. The further abstract syntax given in Figure 6 extends that in Figure 4 and enables us to write basic specifications resembling those in OBJ (as exemplified in Figures 1–3).

p : Positive

$\Longrightarrow$

Basic	$\geq$	"constant" Symbol_0 Relator Term **|**
		"operation" Symbol_p ":" $\mathsf{Functionality}_p$;
Clause	$\geq$	Clause ";" Clause **|** Symbol_p ":" $\mathsf{Functionality}_p$;
Formula	$\geq$	Formula ";" Formula ;
Relator	$\geq$	"$\geq$" **|** ":-" ;
Terms_2	$\geq$	Term $^{\text{"2"}}$;
$\mathsf{Functionality}_p$	$\geq$	Terms_p "$\rightarrow$" Term **|**
		Terms_p "$\rightsquigarrow$" Term **|** Terms_p "$\Rightarrow$" Term **|**
		$\mathsf{Attribute}_p$ **|** $\mathsf{Functionality}_p$ $\mathsf{Functionality}_p$;
$\mathsf{Attribute}_2$	$\geq$	"associative" **|** "commutative" **|**
		"idempotent" **|** "unit" Term ;
$\mathsf{Attribute}_p$	$\geq$	"strict" **|** "defined" **|** "elementary"

Figure 6: Abstract Syntax for Basic Specifications

Now, basic unified specifications look quite nice—to the author, at least—but what is their semantics? Let us consider how to reduce them to canonical specifications.

The symbol ";" stands for conjunction in clauses and formulae. The relators "$\geq$" and ":-" stand for the reversals of the relations '$\leq$' and ':', respectively. It is straightforward to reduce any clause using these constructs to a combination of canonical (Horn) clauses; we omit the details here.

The construct 'constant S R T' merely abbreviates the combination of the declaration 'constant S' and the clause 'S R T'. Likewise, 'operation S : F' abbreviates the combination of 'operation S' and the clause abbreviation 'S : F', where F is a "functionality". Thus what appear to be order-sorted signature declarations are really abbreviations for combinations of (unsorted) unified signature declarations and clauses.

There are three main forms of functionality, concerned with so-called "total", "partial", and "general" operations. Total and partial functionalities may be explained in terms of general functionalities and "attributes", which we consider first.

The functionality '$S : T_1, \ldots, T_p \Rightarrow T$' abbreviates the clause (actually, formula) '$S(T_1, \ldots, T_p) \leq T$'. The monotonicity of all operations gives as a consequence that applying the operation S to any choices (or elements) included in the T_i always gives a result included in T.

Any attributes specified along with such a general functionality enhance it as follows (assuming all arguments are included in the T_i):

- 'strict' asserts that when any argument is 'nothing', the result is 'nothing'.
- 'defined' asserts that when the result is 'nothing', at least one argument must be 'nothing'.
- 'elementary' asserts that when all the arguments are elements, the result is either an element or 'nothing', and, moreover, that the operation is "linear" (i.e., "additive"), preserving '_ | _' and '_&_' in each argument separately.
- 'associative', 'commutative', 'idempotent', and 'unit T'' assert the obvious properties for binary operations. (By the way, 'T^2' abbreviates 'T, T'.)

Now it is easy to explain the "total" and "partial" functionalities:

- '$S : T_1, \ldots, T_p \rightarrow T$' abbreviates
  '$S : T_1, \ldots, T_p \Rightarrow T$ strict defined elementary' (the combination of

'defined' and 'elementary' implies that elements get mapped to elements, hence choices that include elements get mapped to choices that include elements).

- '$S : T_1, \ldots, T_p \rightsquigarrow T$' abbreviates '$S : T_1, \ldots, T_p \Rightarrow T$ strict elementary' (so elements may get mapped to 'nothing').

In practice, it is convenient to extend almost all operations from elements to choices by using "total" or "partial" functionalities. The "general" functionalities are needed only for non-strict operations (such as 'if_then_else_') and for operations that are non-linear (such as the list operation '_(of_)' in its second argument).

As in order-sorted algebras, an operation may have more than one functionality: the clause '$S : F_1\ F_2$' abbreviates the conjunction '$S : F_1'$; $S : F_2'$', where F_1' and F_2' each contain all the attributes of F_1 and F_2, and together contain all their total, partial, and general functionalities. It is claimed that any clause of the form '$S : F$' can be reduced to a conjunction of clauses not involving functionalities.

Put together (and formally defined!) the above reductions serve to convert basic specifications into canonical specifications, thereby providing a "transformational semantics" for basic specifications.

3 Action Semantics

This section explains the general idea of Action Semantics, and gives a simple illustrative example of its use. The necessary pieces of Action Notation are introduced formally, but their intended (operational) interpretation is merely indicated informally. Attention is drawn to the exploitation of unified algebras in Action Notation. This is not a "tutorial" on how best to formulate action semantic descriptions—in fact the example given is not optimal with regard to pragmatic aspects such as modularity and modifiability.

Note that the version of Action Notation used here is somewhat tentative: it has not been "polished", nor has it yet been sufficiently tested on large-scale examples. (Previous versions have been shown adequate for the action semantics of a variety of programming languages, including PASCAL, JOYCE, STANDARD ML, BETA, CCS and CSP.)

3.1 Semantics

As mentioned in the Introduction, an action semantics for a programming language is a compositional mapping from abstract syntactic entities to abstract semantic entities called "actions". Thus it is like a denotational semantics, except that the denotations of constructs are (in general) actions, rather than higher-order functions on Scott domains.

The aim of Action Semantics is to obtain better pragmatic qualities than those of Denotational Semantics—without sacrificing formality! For a critique of the pragmatic qualities of denotational semantic descriptions, together with motivation for the use of Action Semantics, see [17]. Here, let us take the desirability of action semantic descriptions for granted, and proceed to examine their form.

For illustration, we give an action semantics for a simple fragment of an imperative programming language. No claims are made for the practicality of this language: it has been chosen purely to allow an uncluttered demonstration of the use of Action Notation.

The abstract syntax of the language is specified by the (context-free) grammar in Figure 7. Abstract syntax is concerned only with the compo-

Statement ≥	Statement ";" Statement |	
	Identifier ":=" Expression |	
	"if" Expression "then" Statement "else" Statement |	
	"while" Expression "do" Statement |	
	"result" Expression	;
Expression ≥	Numeral | Identifier | Statement |	
	Expression Operator Expression	;
Operator ≥	"or" | "and" | "=" | "+" | "−" | "*"	

Figure 7: Abstract Syntax of an Illustrative Programming Language

sitional structure of programs and their component "phrases", in contrast to concrete syntax, which is concerned (also) with the representation of programs by strings of characters. For specifying abstract syntax, it is convenient to use context-free grammars (here, we use the same variant of BNF as in Section 2). The abstract syntactic entities may be thought of as derivation trees for strings of terminal symbols that can be generated by the grammar; then the nonterminal symbols stand for sets (or choices!) of trees. Ambiguity of grammars is irrelevant for abstract syntax—in fact, blatantly ambiguous grammars often facilitate semantic description, as is the case with our illustrative programming language.

The terminal symbols of the given grammar suggest familiar concrete symbols; but there is no formal connection with concrete syntax (a mapping from concrete to abstract syntax could be given separately). Notice that the nonterminal symbols 'Numeral' and 'Identifier' are left unspecified.

Notation for semantic functions, mapping abstract syntactic entities to their denotations, is introduced in Figure 8. For convenience of no-

execute_	: Statement ⇒ Action(giving nothing)(escaping value)
evaluate_	: Expression ⇒ Action(giving value)
operation_	: Operator ⇒ Data(taking value_1 | value_2)(yielding Value)
valuation_	: Numeral → Number
id_	: Identifier → Token

Figure 8: Semantic Functions

tation, let us treat abstract syntax as a unified algebra, and semantic functions as operations of a unified algebra that encompasses both syntactic and semantic entities.

The intended interpretation of the standard semantic entities 'Action(...)' and 'Data(...)', which are part of Action Notation, is explained in the following sections.

Some special semantic entities (not provided by Action Notation) are introduced in Figure 9. The constant 'Number' is not fully specified. All that we need to know is that it includes the denotations of numerals, and that there are various operations on numbers, in particular '_is_', which tests for identity. The constant 'Value' combines various classifications that would be distinguished in a more careful specification: the results of expression evaluations, and the operands and results of operators. The operations 'Bindable_' and 'Storable_' indicate the classifications of data that may be bound to identifiers, respectively assigned to variable identifiers.

```
constant    Number ≤ Data
operation   _is_ : Number² → Truth-Value
...
operation   product : Number² ⤳ Number
constant    Value = Truth-Value | Number
Bindable(id Identifier) = Cell
Storable(Cell) = Number
```

Figure 9: Special Semantic Entities

The semantic functions are defined inductively, by "semantic equations", as in Denotational Semantics. (The equations may be regarded as algebraic equations, but their "well-foundedness" is essential). In general, each equation corresponds to a homomorphism condition: the denotation of a particular kind of compound phrase is equated with a particular composition of the denotations of sub-phrases; the denotation of a primitive phrase is identified directly with a semantic entity.

The semantic equations for the statements of our illustrative language are given in Figure 10. For now, merely observe the form of the equations: the action notation used in the right-hand-sides—which is entirely formal!—has yet to be explained (although it is hoped that at least some of its intended interpretation is suggested by the words used).

Further semantic equations, defining the denotations of expressions and operators, are given in Figure 11. The first three equations are not in the usual inductive form, because numerals, identifiers, and statements are regarded as special sub-classifications of expressions, rather than as components of expressions. Formally, the denotation of a statement S is the *pair* of actions execute S, evaluate S.

The semantic functions for numerals and identifiers are left unspecified here, as they do not involve actions at all, and anyway, their compositional structure has not been specified by the given abstract syntax.

So much for the basic structure of action semantic descriptions (which could be made more evident by use of explicit modules, as shown in [15]). It remains to introduce Action Notation, and to explain its intended interpretation.

```
execute⟦ S₁ ";" S₂ ⟧ = execute S₁ and then execute S₂
execute⟦ I ":=" E ⟧ =
    | | obtain a cell from bound(id I)
    | and
    | | evaluate E then obtain a number from the value
    then
    | store the number in the cell
execute⟦ "if" E "then" S₁ "else" S₂ ⟧ =
    | evaluate E then obtain a truth-value from the value
    then
    | | check the truth-value and then execute S₁
    | or
    | | check not the truth-value and then execute S₂
execute⟦ "while" E "do" S ⟧ =
    | □
    where □ =
    | | evaluate E then obtain a truth-value from the value
    | then
    | | | check the truth-value and then execute S and then □
    | | or
    | | | check not the truth-value
execute⟦ "result" E ⟧ = evaluate E then (escape(taking value))
```

Figure 10: Semantic Equations for Statements

3.2 Actions

Actions are semantic entities that have a "computational", rather than "mathematical", essence: they can be *performed* so as to *process information*. For the moment, we need not be concerned about what kind of information is processed by (performances of) actions.

When an action is performed, information is usually processed *gradually*, rather than instantaneously. Particular performances of an action may be classified as follows:

- the performance *never* terminates: it is said to *diverge*;

N : Numeral $\Longrightarrow$

evaluate N = obtain a value from valuation N

I : Identifier $\Longrightarrow$

evaluate I = obtain a value from
| bound(id I)(yielding value) |
| stored(bound(id I)(yielding Cell))

S : Statement $\Longrightarrow$

evaluate S = | execute S then irrevocably fail
trap
| complete(taking value)

evaluate⟦ E_1 O E_2 ⟧ =

| | evaluate E_1 then obtain a value$_1$ from the value
| and
| | evaluate E_2 then obtain a value$_2$ from the value
then
| obtain a value from operation O

operation⟦ "or" ⟧ = disjunction(the value$_1$(yielding Truth-Value),
the value$_2$(yielding Truth-Value))

...

operation⟦ "*" ⟧ = product(the value$_1$(yielding Number),
the value$_2$(yielding Number))

Figure 11: Semantic Equations for Expressions and Operators

- the performance terminates *normally*: it is said to *complete*;
- the performance terminates *abnormally*: it is said to *escape*;
- the performance terminates *prematurely*: it is said to *fail*.

Obviously, diverging actions are required to represent the semantics of programs that get into infinite loops. (Note that not all such programs are useless: operating systems and traffic-light controllers may do significant information processing without ever terminating.)

Completing actions represent the semantics of ordinary programs that process a finite amount of information and then terminate.

Escaping actions are needed to allow the performance of a part of an action to avoid the performance of other parts that would normally follow it, but resuming normal performance later.

Two kinds of failing actions are distinguished, according to whether the failure occurs "immediately", or after some "irrevocable" information processing. Immediate failure indicates the lack of an outcome of a performance, and is disregarded in a nondeterministic choice; actions that may fail immediately, depending on the information given to them, are useful as "guards" on choices. Irrevocable failure is a definite outcome, useful for representing the semantics of programs that are forced to stop because of an "error"—numerical overflow, for instance.

Some notation for actions is introduced in Figure 12. (Perhaps the reader objects to the large number of operations and constants in Action Notation. In practice, however, it does seem to be best to use different notation for representing different operational concepts. Even in conventional Denotational Semantics, one usually introduces "auxiliary" notation, rather than using the pure λ-notation.)

The intended interpretation of the introduced notation is explained as follows.

The constant 'Action' classifies *all* actions—not just those that can be expressed using the given constants and operations (which by themselves are rather trivial).

The constant 'fail' is a synonym for 'nothing'; likewise, '_or_' is a synonym for '_ | _' (restricted to actions). These special action symbols are introduced because they are a bit more suggestive that the general symbols, and because they have been used in previous versions of Action Notation. Anyway, 'fail' does what it says, immediately. The action 'A_1 or A_2' may be regarded as a "tentative" choice between performing A_1 and A_2, with "back-tracking" if the chosen action fails immediately. The action 'irrevocably A' performs A, but cannot fail immediately (even if A is 'fail'); hence "committed" nondeterministic choice can be expressed by '(irrevocably A_1)or(irrevocably A_2)'.

The action 'A_1 and A_2' performs A_1 and A_2 together, with arbitrary (perhaps "unfair") interleaving of the performances of their *indivisible* sub-actions; 'indivisibly A' performs any action A indivisibly, protecting the sub-actions of A from interleaving with other actions. (Primitive actions may be assumed to be indivisible unless otherwise stated.) The

constant	Action
constant	fail $\leq$ Action
operation	_or_ : Action^2 $\Rightarrow$ Action associative commutative idempotent unit(fail)
constant	irrevocably_ : Action $\Rightarrow$ Action
operation	_and_ : Action^2 $\Rightarrow$ Action associative commutative
operation	indivisibly_ : Action $\longrightarrow$ Action
operation	_and then_ : Action^2 $\Rightarrow$ Action associative commutative
constant	complete : Action
operation	_then_ : Action^2 $\Rightarrow$ Action associative unit(complete)
constant	escape : Action
operation	_trap_ : Action^2 $\Rightarrow$ Action associative unit(escape)
constant	$\Box$: Action
operation	_where $\Box$ =_ : Action^2 $\Rightarrow$ Action

Figure 12: Some Action Notation

action 'A_1 and then A_2' is the specialization of 'A_1 and A_2' to the interleaving where all of A_1 is performed before any of A_2.

Next, the action 'complete' does just what it says. The action 'A_1 then A_2' performs A_1, followed by A_2 if the performance of A_1 completes; if A_1 diverges, fails, or escapes, A_2 is not performed.

The action 'escape' does what it says. The action 'A_1 trap A_2' performs A_1, followed by A_2 if the performance of A_1 escapes; if A_1 diverges, fails, or completes, A_2 is not performed. Notice the symmetry between completing and escaping (however, an untrapped escape always terminates an interleaving, whereas completion need not).

The action '$\Box$' is a dummy action: whenever it is encountered during the performance of A_1 in 'A_1 where $\Box = A_2$', the action A_2 is performed instead. Essentially, 'A_1 where $\Box = A_2$' abbreviates the (possibly infinite) action obtained by repeatedly replacing occurrences of '$\Box$' in A_1 by A_2 (more precisely, by 'complete then A_2 then complete', to ensure that there is always a "first" step of an action). Of course, '$\Box$' is not assumed to be indivisible.

3.3 Information

Let us now consider the *information* processed by actions. It consists of "organized data".

It is a simple matter to allow an action to compute a particular data entity: just make the action into an operation, and apply it to a term that denotes the required data entity. But then, of course, the *same* data entity gets computed by every performance of the action, which is not much use. What we want is to have terms denoting data that *depends* on some given information. So let us consider a dependent data entity to be an entity that can be *evaluated*, with some information, to yield a data entity. (Ordinary independent data, such as truth-values and numbers, always evaluates to itself. Actions themselves may be regarded as dependent data, but usually they are left unevaluated, with their dependent data components evaluated only when the action gets *performed*.)

Evaluation of dependent data is, in contrast to performance of actions, essentially "mathematical", rather than "computational": evaluation does not involve any *changes* to information, merely *reference* to the given information. Evaluation of dependent data cannot diverge, fail, or escape. The evaluation of dependent data may yield an element of data, but it may also yield a multiple choice, or even the vacuous choice, 'nothing'.

The basic dependent data entities are simply references to particular components of the information given to their evaluation. In general, compound dependent data entities are formed by applying ordinary data operations to dependent data arguments; the evaluation of such a compound entity yields the result of applying the operation to the data yielded by evaluating the arguments. Notice that non-strict operations (such as 'if_then_else_') may ignore a 'nothing' yielded by the evaluation of an argument.

As data may depend on information, the data computed by an action may depend on the information *received* by the action, i.e., the "current" information. The data computed by an action may be incorporated into the information *produced* by the action.

We may classify information according to the way it is propagated by action combinators. The following description should give the main idea, which underlies the design of Action Notation.

- *Transient* information produced by an action is received only at the start of the "immediately-following" action, unless the latter action explicitly re-produces the information. Such information is used to represent intermediate results in computations—values of sub-expressions, for instance.
- *Scoped* information produced by an action is received throughout an immediately-following action, except where explicitly overridden. It is used to represent bindings established by declarations.
- *Stable* information produced by an action is received by all the following actions, until explicitly overridden. It is used to represent the assignment of values to variables.
- *Permanent* information produced by an action is like stable information, but can never be overridden. It is used to represent communication histories.

Of course it is possible to use one kind of information to represent another kind—e.g., using transient information to represent the "state" of assignments to variables, making sure that all actions pass along the state with the appropriate changes. But such abuse of action notation tends to give poor pragmatic qualities in action semantic descriptions (since they start to resemble conventional denotational semantic descriptions!).

The various kinds of information are what may be called "orthogonal": actions process simple aggregations of them, and the processing of each kind of information may be considered separately. Focusing on one kind of information at a time gives what are called the "facets" of actions. The *functional facet* is concerned only with transient information; the *declarative facet*, with scoped information; the *imperative facet*, with stable information; and the *communicative facet*, with permanent information.

The remaining action notation used in the semantics of our illustrative programming language is introduced in Figure 13.

```
constant     Data
operation    _(yielding_) : Data² ⇒ Data
operation    check_ : Data(yielding Truth-Value) ⇒ Action

constant     Name ≤ Data
operation    Nameable_ : Name ⇒ Data
operation    the_ : Name ⇒ Data(yielding Nameable(Name))
operation    obtain an_from_ : Name, Data(yielding Nameable(Name))
                                   ⇒ Action
operation    _(taking_) : Action, Name ⇒ Action
operation    _(giving_) : Action, Name ⇒ Action
operation    _(escaping_) : Action, Name ⇒ Action

constant     Token ≤ Data
operation    Bindable_ : Token ⇒ Data
operation    bound_ : Data(yielding Token) ⇒
                         Data(yielding Bindable(Token))

constant     Cell ≤ Data
operation    Storable_ : Cell ⇒ Data
operation    stored_ : Data(yielding Cell) ⇒
                         Data(yielding Storable(Cell))
operation    store_in_ : Data(yielding Storable(Cell)),
                           Data(yielding Cell) ⇒ Action
```

Figure 13: Some More Action Notation

The constant 'Data' classifies dependent data as well as ordinary data. Evaluation of 'D_1(yielding D_2)' yields the agreement of the data yielded by D_1 and by D_2. (This is only of interest when D_1 or D_2 is dependent data; otherwise, agreement '_&_' could be used directly.)

The action 'check t' completes, provided t yields the element 'true'; it fails immediately if t yields 'false', or 'nothing'. Notice that '(check t then A_1) or (check not t then A_2)' expresses ordinary conditional choice between A_1 and A_2 (at least when t always yields an element of 'Truth-Value').

Not indicated in Figure 13 is the extension of all ordinary data operations to dependent data: if o stands for an operation (with n arguments) and $D_1, \ldots, D_n$ denote arbitrary dependent data, the term '$o(D_1,\ldots,D_n)$' denotes the dependent data that evaluates the D_i and applies o to the data yielded.

Now for some notation specifically concerned with the *functional* facet of actions. It is convenient to use symbolic *names* (rather then the positional notation sometimes used in functional programs) for referring to particular components of transient information. The constant 'Name' classifies all names. The operation 'Nameable_' maps each name to the classification of data to which it may refer—notice the exploitation of unified algebras here. For our illustrative semantics, the only names needed are 'cell', 'number', 'truth-value', and 'value' (with optional subscripts). For brevity, the formal introduction of these names is omitted here; the corresponding "nameables" are evident.

Let n denote a name. Then 'the n' yields the data named n in the given naming—or just 'nothing', if there is no such data.

When D denotes (dependent) data and N denotes a *choice* of names, the term 'D(taking N)' yields whatever D yields with the given naming restricted to names included in the choice N.

The action 'obtain an n from D' completes, producing the naming of the data yielded by evaluating 'D(yielding Nameable n)'—unless that is 'nothing', in which case the action fails immediately. Note that the naming of a multiple choice is different to the choice between the namings of its elements.

When N is a *choice* of names, the action 'A(taking N)' performs 'complete(taking N) then A', and the action 'A(giving N)' performs 'A then (complete(taking N))'. Note that these actions are equivalent to A when A refers to, respectively normally produces data with names included in N. Similarly, 'A(escaping N)' performs 'A trap (escape(taking N))'. This leaves 'complete(taking N)' to be explained: it merely reproduces the restriction of the received naming to the names included in the choice N.

Now for the *declarative* facet of actions, concerned with scoped information. The elements of 'Token' may be bound to data, according to the operation 'Bindable_'. Binding actions are quite interesting, but out of the scope (!) of this paper: all we require here is to refer to a received binding for a token k, which is expressed by 'bound k'.

Similarly, for the *imperative* facet, concerned with stable information, we have 'Cell' and 'Storable_'; reference to the current data stored in a cell c is expressed by 'stored c'. The action 'store d in c' assigns data d to a cell c; it is irrevocable, as well as indivisible.

It remains to explain how actions deal with information that consists of namings, bindings, and storage all together. Note straight away that dependent data arguments in an action (such as t in 'check t') are always evaluated with all the received information.

Storage is rather obvious: the only action (here) that changes the storage is 'store_in_'. All other actions leave the storage that they receive unchanged. Note that interleaving 'A_1 and A_2' lets A_1 and A_2 influence each other's performance by means of changes to storage. The irrevocability of changes ensures that the choice of interleaving never has to be made tentatively.

Binding is trivial, in the absence of the binding actions: all actions receive the same bindings, and it is unnecessary to consider the production of bindings at all.

The treatment of namings is as follows:

The actions 'complete' and 'escape' both re-produce whatever naming their performances receive. But 'fail' produces the null naming, as do 'check T' and 'store d in c'.

The actions 'irrevocably A' and 'indivisibly A' are "transparent" with regard to the namings received and produced.

When 'A_1 or A_2' is performed, the chosen alternative receives and produces the same namings as the combined action.

'A_1 and A_2' performs A_1 and A_2 *separately*, as regards namings: A_1 and A_2 both receive the same naming as the combined action, and (if they both complete) the naming produced by the combined action is just the combination of the namings they produce. Thus the interleaving of the performances of A_1 and A_2 does *not* let intermediate namings produced by A_1 be referred to by A_2. If an escape occurs, the naming it produces is the naming produced by the combined action—the (intermediate) naming produced by the other action is ignored. 'A_1 and then A_2' is similar.

'A_1 then A_2' performs A_1 with the received naming, and performs A_2 (if at all) with the naming produced by A_1. This corresponds to "functional composition". 'A_1 trap A_2' is analogous.

The namings received and produced by 'A_1 where $\square = A_2$' are those of A_1, taking into account the replacements of '$\square$' in A_1 by A_2. Thus A_2 does not in general receive the same naming as the combined action.

The main effect of the treatment of namings described above is that the received naming is always propagated to both arguments of '_or_', '_and_', and '_and then_', but only to the first argument of '_then_', '_trap_', and '_where $\square$ =_'.

The reader might now find it refreshing to re-examine the semantic equations for the illustrative programming language.

4 Conclusion

We have considered the frameworks of Unified Algebras and Action Semantics separately. Let us conclude by summarizing the relation between them.

Unified Algebras make intensive use of the choice operation, which corresponds to union of classifications: not only is it provided for direct use in specifications (recall the sort equations for 'Natural', etc.) but also it underlies the inclusion partial order, which is preserved by all the other operations.

Action Semantics makes direct use of choices between actions for representing nondeterministic (and deterministic) choice; also, implementation-dependent order of execution is expressed by '_and_', which involves choice of interleaving. Choices are useful in connection with restricting the names of data received and produced by the functional facet of actions (which is analogous to the restriction of list components by the operation '_(of_)' in unified algebras). Finally, choice corresponds exactly to the "alternatively" in BNF productions, which allows the use of unified algebras for abstract syntax as well as for semantic entities.

Action Notation exploits the generality of Unified Algebras to use the same notation for operations on classifications of actions and on particular actions, e.g., the operation '_(taking_)'. Operations such as 'Nameable_' map elements to classifications, avoiding the clumsy indexed families of sorts needed in earlier formulations of Action Notation.

Thus it can be seen that Unified Algebras have made a significant contribution to Action Semantics. But of course it was Action Semantics that came first, and by embracing nondeterminism (rather than avoiding it) prepared the ground for Unified Algebras.

Acknowledgments. Inspiration for Unified Algebras came from the work of Joseph Goguen and José Meseguer, Gert Smolka, Hassan Aït-Kaci, and Bill Wadge. David Watt collaborated on the development of Action Semantics; the conceptual analysis of programming languages exploited in Action Semantics was developed (for use in Denotational Semantics) by Christopher Strachey. Thanks to the organizers of STACS'89 for the invitation to write a paper on an unspecified topic.

References

[1] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In *Proc. Winter School on Abstract Software Specifications (Copenhagen, 1979)*. Springer-Verlag (*LNCS* 86), 1980.

[2] *The Concise Oxford dictionary of current English.* Oxford University Press, sixth edition, 1976.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[4] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. POPL'85.* ACM, 1985.

[5] J. A. Goguen. Order sorted algebra. Semantics and Theory of Computation Report 14, UCLA Computer Science Dept., 1978.

[6] J. A. Goguen. Higher order functions considered unnecessary for higher order programming. Technical Report SRI-CSL-88-1, Computer Science Lab., SRI International, Jan. 1988.

[7] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Proc. Logics of Programming Workshop*, pages 221–256. Springer-Verlag (*LNCS* 164), 1984.

[8] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for computer science. Report CSLI-85-30, CSLI, Stanford University, 1985.

[9] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Partial and overloaded operators, errors and inheritance. Technical report, Computer Science Lab., SRI International, 1987.

[10] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, Volume IV*. Prentice-Hall, 1978.

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

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

[13] G. Grätzer. *Lattice Theory: First Concepts and Distributive Lattices*. W. H. Freeman & Co., 1971.

[14] P. D. Mosses. Unified algebras and modules. In *Proc. POPL'89*. ACM. To appear.

[15] P. D. Mosses. The modularity of action semantics. Internal Report DAIMI IR–75, Computer Science Dept., Aarhus University, 1988. Revised version of a paper presented at a CSLI Workshop on Semantic Issues in Human and Computer Languages, Half Moon Bay, California, March 1987. Available from the author. To appear in *SDF Benchmark Series in Computational Linguistics – Workshop II*, MIT Press.

[16] P. D. Mosses. Unified algebras and institutions (extended abstract). Internal Report DAIMI IR–83, Computer Science Dept., Aarhus University, 1988. Available from the author; full version in preparation.

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

[18] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[19] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, 1986.

[20] D. S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.

[21] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata, Microwave Research Institute Symposia Series Volume 21.* Polytechnic Institute of Brooklyn, 1971.

[22] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. Order-sorted equational computation. SEKI Report SR–87–14, FB Informatik, Universität Kaiserslautern, 1987.

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

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

PB – 272 P. D. Mosses: Unified Algebras and Action Semantics