ISSN 0105-8517

ABSTRACT SEMANTIC ALGEBRAS!

by

Peter Mosses

DAIMI PB-145

July 1982

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

ABSTRACT

A new approach to the formal description of programming language semantics is described and illustrated. "Abstract semantic algebras" are just algebraically-specified abstract data types whose operations correspond to fundamental concepts of programming languages. The values of abstract semantic algebras are taken as meanings of programs in Denotational (or Initial Algebra) Semantics, instead of using Scott domains. This leads to semantic descriptions that clearly exhibit the underlying conceptual analysis, and which are rather easy to modify and extend. Some basic abstract semantic algebras corresponding to fundamental concepts of programming languages are given; they could be used in the description of many different programming languages.

Note. This paper is to appear in the Proceedings of the IFIP TC2 Working Conference on Formal Description of Programming Concepts II, Garmisch-Partenkirchen, June 1982, to be published by North-Holland Publishing Company, Amsterdam. It is made available as a preprint for the personal use of the recipient. References should cite the Proceedings (only).

INTRODUCTION

The work reported here is based on a combination of ideas from Denotational Semantics[20,33], Initial Algebra Semantics[1] and the algebraic specification of abstract data types[2]. The reader is assumed to be familiar with these topics.

An Abstract Semantic Algebra *(ASA)* is a special sort of algebraically-specified abstract data type *(ADT)*. Its operations represent fundamental concepts of programming languages, such as particular forms of sequence control (*e.g.* sequencing, conditionals, iterations) and data control (*e.g.* storage, bindings, scope rules). The values of ASAs are *actions* which, when executed, may consume and produce data, establish and use bindings, and have side-effects. The operations on actions are specified by algebraic axioms that give their essential properties — it is not necessary to model them using auxiliary objects like states and environments. There are, in general, many different possible models of the axioms of an ASA specification, including the (discrete) initial one, which is taken as its meaning.

Some basic ASAs are given below. They correspond closely to fundamental concepts of programming languages in the analysis made by the late Christopher Strachey[32], which is often followed in denotational descriptions[21,15]. A small library of basic ASAs has been developed, incorporating many of the concepts to be found in present-day programming languages (apart from type-checking and concurrency).

The proposed approach to formal semantics is to give denotational descriptions, taking denotations of program phrases to be elements of ASAs (instead of Scott domains[29]). Now, a denotational description consists of three main parts, specifying Abstract Syntax, Semantic Values and Semantic Functions. With the proposed approach, abstract syntax is specified as usual, in a BNF-like notation. Semantic values are specified partly in terms of some ordinary abstract data types, giving the basic values of the programming language (numbers, truth-values, *etc.*); partly as a selection of basic ASAs, giving actions consuming, producing and binding the specified values. Finally, semantic functions are specified using the familiar semantic equations. However, denotations of program phrases are now expressed (as actions) using the operations of the selected ASAs, rather than (as functions) in λ-notation. This ensures

that semantic equations exhibit how the described programming language has been analysed in terms of fundamental concepts. (Alternatively, they may be regarded as showing how the programming language may be *synthesised* by taking combinations of fundamental concepts and then coating them with "syntactic sugar".)

In Denotational Semantics, the BNF-like specifications of abstract syntax are generally interpreted as abbreviations for domain equations. However, they can alternatively be regarded as abbreviations for signatures of classes of algebras, and abstract syntax identified with initial algebras in these classes. Then a set of semantic equations (provided that these are "homomorphic") makes the semantic values into a target algebra for an initial algebra semantics[1], with the semantic function being uniquely determined as a homomorphism from the syntactic initial algebra to this target algebra. When the semantic values are ASAs, a set of semantic equations gives a "derivor" in the terminology of the ADJ group[2].

This, then, is the essence of the proposed approach: an abstract syntax is an initial algebra, with the signature being specified in BNF; semantic values are taken from standard ADTs and ASAs; and a semantic function is the unique homomorphism determined by a target algebra, which is specified by a set of semantic equations. The resulting specifications look quite like ordinary Denotational Semantics (apart from the absence of λs!) but their interpretation is firmly algebraic.

The main aim here is to obtain semantic descriptions that: (i) exhibit a semantic analysis based on fundamental concepts of programming languages; (ii) are easily modifiable and extendable, and (iii) make use of standard, language-independent modules. These pragmatic aspects are of particular concern when semantic descriptions are to be used in connection with language design, standards, implementations and teaching.

Related Work
Standard Denotational Semantics, founded by Scott and Strachey[20,30], has been used for describing quite a few current programming languages[*e.g.*21,17]. However, it is not ideal with respect to the pragmatic aspects mentioned above: (i) Fundamental concepts of programming languages are not exhibited, instead they are encoded in λ-notation and modelled by manipulation of higher-order functions. (ii) Semantic equations are committed to particular "styles" of model, *e.g.* the "direct" or "continuation" styles. They require extensive (albeit systematic[31]) rewriting when the described language is to be extended with features that cannot be modelled in the chosen style, *e.g.* jumps in the direct style, concurrency in the continuations style. (iii) There is no sharing of standard modules between descriptions of different languages, even when they have many features in common.

One may obtain a standard denotational description from one based on ASAs by choosing a model (using Scott domains) for the axioms of the ASA specifications. The resulting description differs from the usual standard ones in that the semantic equations are not committed to the chosen model, thanks to the abstraction of operations corresponding to fundamental concepts.

The ADJ group recognised the algebraic structure implicit in Denotational Semantics, and identified abstract syntax with initial algebras, obtaining Initial Algebra Semantics[1]. The approach proposed here is basically Initial Algebra Semantics, with target algebras taken to be abstract data types (derived from ASAs) instead of "concrete" data types (based on domains). The specifications given below differ in style from those of [1] in that BNF is used to specify signatures here, and semantic equations are used to specify the construction of target algebras from semantic values.

Wand[34] suggested using abstract data types in semantic descriptions. In contrast to the approach here, his "semantic functions" were non-homomorphic, they were just operations of abstract data types that combined syntax and semantics. He also used auxiliary objects, such as states, rather than axiomatising sequence and data control directly. Nevertheless, Wand's work has provided much inspiration for the present approach.

Goguen and Parsaye-Ghomi[12] essentially followed Wand, but they introduced a modular, hierarchical structure into semantic descriptions, using the specification language OBJ[10]. They also used auxiliary objects such as states and environments, but based them on reusable standard modules (*e.g.* LIST, ARRAY). Their example semantic description corresponds quite closely to a standard denotational description, with abstract data types taking the place of domains — but procedures are treated syntactically, rather than denotationally.

However, modular structure alone seems not to be enough to ensure good modifiability of semantic descriptions: their semantic equations are just as much committed to a particular style of model as those of ordinary domain-based denotational descriptions, and would need extensive rewriting to accommodate a continuation-style model, for instance. In the approach proposed here, good modifiability is claimed to come from the use of operations that, in corresponding directly to fundamental (independent) concepts, do not presume any particular model. (The presentation of basic ASAs below would undoubtedly be improved by the use of a specification language like OBJ, allowing the hierarchical structure of the modules to be expressed formally.)

Broy and Wirsing[6] gave an abstract data type that corresponds rather closely to some of the basic ASAs presented below, and made a careful investigation of models. There are differences in the algebraic foundations — it is not clear to this author how significant these differences are. Broy and Wirsing seem to regard their abstract data type as a programming language in itself, rather than as a target algebra for initial algebra semantics.

The work of Gaudel[9] and Pair[25] is, like the approach proposed here, based on abstract data types and Initial Algebra Semantics. The main difference is in the choice of abstract data types, especially for dealing with procedural abstractions: their operations seem to correspond

more closely to particular implementation techniques than do the basic ASAs given below. Again, there are also some differences in the algebraic foundations.

It should be mentioned that numerous authors of denotational descriptions have made use of operations similar to those of the basic ASAs below. Usually, these operations have been introduced as *ad hoc* combinators, abbreviating complex pieces of λ-notation. However, some general language-independent combinators have been proposed by Raoult and Sethi[26] and by Wand[35]. They are specified as functions on domains, and some of their algebraic properties are proved.

Earlier papers by the author used ASAs in treating compiler correctness[23] and the semantics of binding constructs[24]. The reader is warned that there are substantial differences between the operations of the ASAs specified below and those in the earlier papers. The changes have been made in an attempt to get a better correspondence between operations and fundamental concepts, and to enhance the readability of semantic descriptions using the ASAs. (Perhaps the ASAs given here will also give way to better ones, in their turn.)

Christiansen and Jones[7] make use of the basic ASA for sequence control given here (with minor variations) in their semantics-directed compiler-generator.

Currently, a couple of students at Aarhus are completing a semantic description of Pascal, based on ASAs similar to the ones given below. The description has been partially tested by choosing a standard model and running it on SIS[22]. Although the result of this first major test of the use of ASAs is encouraging, much more experience of using the basic ASAs is needed before one can be sure of their appropriateness.

Plan

The rest of this paper is organised as follows. First, the algebraic foundations of the proposed approach are described, and notation introduced. Then some basic ASAs are specified and motivated. The paper concludes with a small example of a semantic description based on ASAs. The programming language described is M-Lisp, the language used by McCarthy[19] for specifying the operational semantics of Lisp1.5 through an abstract interpreter.

FOUNDATIONS

For the basic algebraic notions and terminology, we mostly follow the ADJ group[1,2] – but note that signatures here are a bit more general, in order to cope with polymorphic operators and implicit coercions. Motivation for the use of algebras in the study of abstract data types may be found in [18,2].

Algebras

Defn. For any set S, an S-sorted *signature* Σ is an $S^* \times S$-indexed family of sets of *operators* $\Sigma_{(s_1,\dots,s_n),s}$. The index $(s_1,\dots,s_n)$ gives the *arity* of operators, the index s gives the *sort* of their result. (It is not assumed that the sets of operators are disjoint.)

Defn. Let Σ be an S-sorted signature. A Σ-*algebra* A consists of an S-indexed family of sets A_s, together with, for each $\sigma \in \Sigma_{(s_1,\dots,s_n),s}$, a function $\sigma_A : (A_{s_1} \times \cdots \times A_{s_n}) \to A_s$ (total). A_s is called the *carrier* of A of sort s, and σ_A is called an *operation* of A named by σ (or a *constant* when $\sigma \in \Sigma_{(),s}$ – then we write just σ_A for $\sigma_A()$).

Defn. Let A, B be S-sorted Σ-algebras. Then a Σ-*homomorphism* $h : A \to B$ consists of an S-indexed family of functions $h_s : A_s \to B_s$, respecting the operations of A:

$$h_s(\sigma_A(a_1,\dots,a_n)) = \sigma_B(h_{s_1}(a_1),\dots,h_{s_n}(a_n))$$

for each $\sigma \in \Sigma_{(s_1,\dots,s_n),s}$, each $(a_1,\dots,a_n) \in A_{s_1} \times \cdots \times A_{s_n}$ (just $h_s(\sigma_A) = \sigma_B$ for constants).

For example, we have a $\{\mathbf{T}\}$-sorted signature *Bool* given by $Bool_{(),\mathbf{T}} = \{\mathbf{ff},\mathbf{tt}\}$, $Bool_{(\mathbf{T}),\mathbf{T}} = \{\neg\}$, and $Bool_{(\mathbf{T},\mathbf{T}),\mathbf{T}} = \{\wedge\}$ (all the other $Bool_{(\mathbf{T},\dots,\mathbf{T}),\mathbf{T}}$ being empty); and a *Bool*-algebra B with $B_{\mathbf{T}} = \{0, 1\}$, given by

$$\begin{array}{lclclcl} \mathbf{ff}_B & = & 0 & \qquad & \neg_B(i) & = & i - 1 \\ \mathbf{tt}_B & = & 1 & & \wedge_B(i,i') & = & i \times i'. \end{array}$$

A class of S-sorted Σ-algebras, together with all the Σ-homomorphisms between them, gives a *category*. Composition of Σ-homomorphisms $f : A \to B$, $g : B \to C$, written $g \cdot f\ A \to C$, is just S-indexed function composition, and the identity Σ-homomorphism on A, written $\mathbf{1}_A$, is given by the S-indexed family of identity functions on the carriers A_s.

Defn. A homomorphism $h : A \to B$ in a category $\mathbf{C}$ of Σ-algebras is called an *isomorphism* iff there is a Σ-homomorphism $h' : B \to A$ in $\mathbf{C}$ with $h' \cdot h = \mathbf{1}_A$ and $h \cdot h' = \mathbf{1}_B$. Two algebras A, B are *isomorphic* iff there exists an isomorphism between them.

Defn. An algebra A is called *initial* in a category $\mathbf{C}$ of Σ-algebras iff for every algebra B in $\mathbf{C}$ there exists a unique Σ-homomorphism $: A \to B$.

Propn. If A and A' are both initial in a category $\mathbf{C}$ of Σ-algebras, then $A \cong A'$. If A is initial in $\mathbf{C}$ and $A \cong A''$, then A'' is also initial in $\mathbf{C}$.

Defn. For any Σ, $\mathbf{Alg}_\Sigma$ is the category of all Σ-algebras.

The following theorem (see [2]) is fundamental:

Theorem. For any Σ, $\mathbf{Alg}_\Sigma$ has an initial algebra, T_Σ.

T_Σ is essentially (up to isomorphism) the S-sorted term algebra over Σ, whose carriers $(T_\Sigma)_s$ are sets of well-formed expressions in the operators of Σ, and whose operations σ_T construct terms '$\sigma(t_1,\dots,t_n)$' from operands $t_1,\dots,t_n$ (the separators '(', ')' and ',' representing symbols not occurring in Σ). The property of Σ-terms ensuring the initiality of T_Σ is that they can be uniquely decomposed.

Actually, a little more care is needed here when the sets of operators $\Sigma_{(s_1,\dots,s_n),s}$ are not disjoint, *e.g.* if $\mathbf{0} \in \Sigma_{(),\mathbf{N}} \cap \Sigma_{(),\mathbf{Z}}$ and $\textbf{is-zero} \in \Sigma_{(\mathbf{N}),\mathbf{T}} \cap \Sigma_{(\mathbf{Z}),\mathbf{T}}$ then the term '**is-zero(0)**' may be constructed (or analysed) in two different ways. This can be fixed by making the sorts of sub-terms explicit, as in '$\sigma(s_1 : t_1,\dots,s_n : t_n)$', *e.g.* '**is-zero(N : 0)**'.

T_Σ is identified with abstract syntax[1], and the initiality of T_Σ in $\mathbf{Alg}_\Sigma$ gives for any *target* Σ-algebra A, a unique homomorphism $: T_\Sigma \to A$, which is a sort-indexed family of semantic functions mapping abstract terms to their interpretations in A. This is Initial Algebra Semantics.

Initial algebras are appropriate not only for abstract syntax, but also as the meanings of equationally-specified abstract data types[2] – also when conditional equations are used[4].

Defn. For any S-sorted signature Σ and S-indexed family of sets of variables X_s (distinct from Σ), $\Sigma(X)$ is as Σ except for $\Sigma(X)_{(),s} = \Sigma_{(),s} \cup X_s$, for each $s \in S$ (*i.e.* the variables are added to Σ as "constants").

Defn. $T_\Sigma(X)$ is the Σ-*term algebra on* X, obtained from $T_{\Sigma(X)}$ by removing the operations named by the variables in X.

Propn. For any Σ-algebra A, for any sort-indexed family of mappings $f_s : X_s \to A_s$, there is a unique homomorphism $f^\# : T_\Sigma(X) \to A$ extending f, *i.e.* with $f^\#(x) = f(x)$ for all variables x.

Defn. A Σ-*equation in* X is a triple, written $t = t' \;\text{--}\; s$, where $t, t' \in (T_\Sigma(X))_s$; or it is a *conditional* Σ-*equation in* X of the form

$$t_1 = t_1' \;\text{--}\; s_1 \;\;\&\;\; \cdots \;\;\&\;\; t_n = t_n' \;\text{--}\; s_n \;\;\Rightarrow\;\; t_0 = t_0' \;\text{--}\; s_0$$

where each $t_i, t_i' \in (T_\Sigma(X))_{s_i}$, for $i = 0$ to n.

Defn. A Σ-algebra A *satisfies* a Σ-equation $t = t' \;\text{--}\; s$ in X iff for every $f: X \to A$, $f^\#(t) = f^\#(t')$; it satisfies a conditional equation

$$t_1 = t_1' \;\text{--}\; s_1 \;\;\&\;\; \cdots \;\;\&\;\; t_n = t_n' \;\text{--}\; s_n \;\;\Rightarrow\;\; t_0 = t_0' \;\text{--}\; s_0$$

in X iff for every $f: X \to A$, whenever $f^\#(t_i) = f^\#(t_i')$ for $i = 1$ to n then $f^\#(t_0) = f^\#(t_0')$. An algebra is said to satisfy a set of equations E iff it satisfies every equation in E.

Defn. For any Σ, for any set of Σ-equations E, $\mathbf{Alg}_{\Sigma,E}$ is the category of all Σ-algebras that satisfy E.

The Initial Algebra Approach to abstract data types[2] takes "the" initial algebra of $\mathbf{Alg}_{\Sigma,E}$ as the meaning of the specification (Σ, E).

Theorem. For any (Σ, E), $\mathbf{Alg}_{\Sigma,E}$ has an initial algebra, $T_{\Sigma,E}$.

$T_{\Sigma,E}$ is essentially the quotient $T_\Sigma / \equiv_E$, where $\equiv_E$ is the least Σ-congruence on T_Σ generated by E.

For example, consider the following *Bool*-equations, where t and t' are variables of sort $\mathbf{T}$:

$$\begin{array}{lclclcl}
\neg(\mathbf{tt}) & = & \mathbf{ff} & \qquad & \wedge(t, \mathbf{tt}) & = & t \\
\neg(\mathbf{ff}) & = & \mathbf{tt} & & \wedge(t, \mathbf{ff}) & = & \mathbf{ff} \\
 & & & & \wedge(t, t') & = & \wedge(t', t).
\end{array}$$

We let the sort of an equation remain implicit when it is unambiguously determined by the given terms. (Of course it can only be $\mathbf{T}$ above, as that is the only sort in the signature *Bool*.)

Specifications

The notation above may be used directly for specifying abstract syntax and abstract data types. However, fully-parenthesised prefix notation for operations can be a bit heavy in practice, and it seems appropriate to allow (formally) infix notation, and even "mixfix" notation, following Rus[28] and Goguen[10]. This is especially the case when specifying abstract syntax of programming languages, as one is able to keep the representation of operators close to the usual concrete syntax. It seems natural then to use *context-free grammars* for presenting signatures along with the notation to be used for expressing their operators.

A context-free grammar G determines an S-sorted signature Σ in the following way. Let G have non-terminals N, terminal symbols T and productions $P \subseteq N \times U^*$, where $U = N \cup T$. (G is allowed to be infinite.) Let each non-terminal $x \in N$ be associated with a sort $s(x) \in S$. Each production $p \in P$ is of the form $x ::= u_0 x_1 u_1 \ldots x_n u_n$, where each $u_i \in T^*$ for $i = 0$ to

$n (\geqslant 0)$; it contributes an operator '$u_0\_u_1\_\cdots\_u_n$' to the set $\Sigma_{(s(x_1),\ldots,s(x_n)),s(x)}$, where '$\_$' is some "place-holder" symbol assumed not to occur in U. The place-holders show how terms are to be written.

Moreover, it is convenient (as well as mnemonically helpful) to take the non-terminals $x \in N$ of G as variables of the associated sorts $s(x) \in S$, allowing subscripts and(or) primes to distinguish different variables of the same sort. (Subscripted non-terminals may be used in productions: they allow easy reference to particular operands, as well as emphasising the context-freeness of the grammar.)

For example (giving the sorts associated with non-terminals in parentheses):

$$(\mathbf{T}) \qquad t \quad ::= \quad \mathbf{ff} \;\mid\; \mathbf{tt} \;\mid\; \neg t \;\mid\; t_1 \wedge t_2$$

specifies the $\{\mathbf{T}\}$-sorted signature Σ with $\Sigma_{(),\mathbf{T}} = \{\mathbf{ff}, \mathbf{tt}\}$, $\Sigma_{(\mathbf{T}),\mathbf{T}} = \{\neg\_\}$, $\Sigma_{(\mathbf{T},\mathbf{T}),\mathbf{T}} = \{\_\wedge\_\}$; and also the variables $X_\mathbf{T} = \{t, t_0, t_1, \ldots, t', t_0', \ldots, t'', \ldots\}$.

Equational axioms are given in the usual way, but using the notation given by the grammar for expressing terms. Parentheses and explicit sorts (written $s : t$) may be used freely to disambiguate terms when G is ambiguous. (One might allow the omission of disambiguating parentheses and sorts when the ambiguity is "safe", *e.g.* when an operation is specified to be associative: if the equation $t \wedge (t' \wedge t'') = (t \wedge t') \wedge t''$ is given, then one could allow the term $t \wedge t' \wedge t''$.)

Now for *coercions*, which are much exploited in the specifications of ASAs. Coercions here are just "invisible" operations, embedding one sort in another. They are useful in giving sorts that are (conceptually) unions of other sorts (sum domains are used in Denotational Semantics for this purpose), and also in relating operators that are defined on different sorts, as in:

$$\begin{array}{llcl}
(\mathbf{N}) & n & ::= & \mathbf{0} \;\mid\; n + \mathbf{1} \\
(\mathbf{Z}) & z & ::= & n \;\mid\; z + \mathbf{1} \;\mid\; z - \mathbf{1}
\end{array}$$

— the coercion $z ::= n$ (*i.e.* $\_ \in \Sigma_{(\mathbf{N}),\mathbf{Z}}$) indicates that $n ::= n + \mathbf{1}$ is to be a restriction of $z ::= z + \mathbf{1}$ to $\mathbf{N}$, with '$\mathbf{0} + \mathbf{1}$' denoting the same $\mathbf{Z}$-value whichever '$\_ + \mathbf{1}$' operation is used.

Goguen[11] and Reynolds[27] treated coercions by partially-ordering sort-sets and augmenting the notions of signature and algebra accordingly. We follow a somewhat more naive approach here, keeping the standard notions of signature and algebra and using algebraic equations for expressing the essential property of coercions, namely that coercions commute with other operations. These equations can be derived systematically from signatures (actually, the grammars for signatures are required to be non-cyclic, so that coercions induce non-identifying partial orderings on sort sets). Unfortunately there is no space here to give the details of this approach to coercions.

Another feature of our specifications of signatures here is the use of indexed families of sorts and operators. Indices are written as superscripts on sorts and variables. For example, one of the basic ASAs has a family of function-like sorts (of "actions") ${}^\sigma\mathbf{A}^\tau$ indexed by their "source" type σ and "target" type τ, where $\sigma, \tau \in$ some set Δ. The composition operator _ ! _ for these actions is restricted (syntactically) to composable operands by:

$$({}^{\sigma}\mathbf{A}^{\tau}) \qquad {}^{\sigma}a^{\tau} \ ::= \ {}^{\sigma}a^{\tau'} \ ! \ {}^{\tau'}a^{\tau}$$

i.e. there is a (σ, τ, τ')-indexed family of composition operators.

Such explicit superscripts on sorts and variables are rather unwieldy, so instead they may be hidden, and subscripts and(or) primes used for referring to an index of a particular variable: the σ-index of a_1 is referred to as σ_1, *etc.*. Then the specification above can be written as

$$({}^{\sigma}\mathbf{A}^{\tau}) \qquad a \ ::= \ a_1 \ ! \ a_2 \qquad -- \ \sigma = \sigma_1; \ \ \tau = \tau_2; \ \ \tau_1 = \sigma_2$$

where the assertions on the right give the indices of the (result) sort of the operator as well as any restrictions on the indices of the operands.

In specifying axioms of ASAs, it is usually appropriate to leave indices entirely implicit: an equation $t = t'$ omitting indices on variables is really an equation *schema*, giving one ordinary equation for each assignment of indices to the variables such that both t and t' are well-formed terms. For example, an axiom like $a \ ! \ a = a$ would give only ${}^{\sigma}a^{\sigma} \ ! \ {}^{\sigma}a^{\sigma} = {}^{\sigma}a^{\sigma}$, since the left-hand side of the axiom is only well-formed if $\sigma = \tau$.

Apart from the notation for signatures and abstract data types described above, the proposed approach uses *semantic equations* for specifying the construction of target algebras. Let T_{Σ} (abstract syntax for Σ) and a Σ'-algebra M' (semantic values) be given, where (in general) $\Sigma \neq \Sigma'$. For each sort s of Σ, let s' be a sort chosen from the sorts of Σ'. Then a set of *homomorphic semantic equations* for Σ gives, for each operator $u_0\_u_1\_\ldots\_u_n \in \Sigma_{(s_1,\ldots,s_n).s}$ an equation of the form

$$\mathscr{F}_s[\![u_0 x_1 u_1 \ldots x_n u_n]\!] = t'(\mathscr{F}_{s_1}[\![x_1]\!], \ldots, \mathscr{F}_{s_n}[\![x_n]\!])$$

where $\mathscr{F}_s : (T_{\Sigma})_s \to M'_{s'}$ is a semantic function for sort s, the x_i are distinct variables of sort s_i for $i = 1$ to n, and $t'(x'_1, \ldots, x'_n)$ is a term in $(T_{\Sigma'}(X'))_{s'}$ where each x'_i is of sort s'_i and $\bigcup_{s' \in S'} X'_{s'} = \{x'_1, \ldots, x'_n\}$. In other words, t' is a composition of the operators of M', with functionality $M'_{s'_1} \times \cdots \times M'_{s'_n} \to M'_{s'}$.

Such a set of homomorphic semantic equations constructs a Σ-algebra M out of M', with $M_s = M'_{s'}$ for each sort s of Σ, the operations of σ_M being the given compositions of the operators of M'. The unique homomorphism $: T_{\Sigma} \to M$ is just the family of functions $\mathscr{F}_s$. (When M' is an abstract data type $T_{\Sigma', E'}$, the transformation form M' to M expressed by the semantic equations is called a *derivor* in the terminology of the ADJ group[2].)

This algebraic interpretation of semantic equations was recognised by the ADJ group[1], who stated that the semantic equations of Denotational Semantics "describe an algebra *and* say that semantics is a homomorphism ... ". However, the semantic equations of standard denotational descriptions in the literature (*e.g.* [15]) don't in fact appear to be entirely homomorphic:

(i) they often seem to give several semantic functions for some sorts, *e.g.* $\mathscr{E}, \mathscr{L}, \mathscr{R} : \mathbf{Exp} \to \ldots$;

(ii) they are sometimes non-compositional, *e.g.*

$$\begin{aligned} \mathscr{E}[\![E_1 - E_2]\!] &= \mathscr{E}[\![E_1 + (-E_2)]\!] \\ \mathscr{C}[\![\mathbf{repeat}\, C]\!] &= \ldots \mathscr{C}[\![C]\!] \ldots \mathscr{C}[\![\mathbf{repeat}\, C]\!] \ldots \end{aligned}$$

(yet still specifying a denotational semantic function);

(iii) the semantics of binding constructs, such as declarations and formal parameters, is given in terms of environments $(\mathbf{Ide} \to \mathbf{D})$, and the creation (or extension) of an environment involves identifiers directly, rather than their denotations, *e.g.*

$$\mathscr{D}[\![\mathbf{const}\, I = E]\!] = \ldots \rho[\epsilon / I] \ldots .$$

How can such semantic equations be regarded as homomorphic?

Concerning (i), one may consider the several functions as components of a single compound (target-tupled) semantic function. The separate names and definitions can be combined, at the expense of inserting a lot of explicit projecting and tupling. (Actually, with the standard $\mathscr{E}, \mathscr{L}, \mathscr{R}$ evaluations on **Exp**, one can usually make do without tuples, as $\mathscr{L}$ and $\mathscr{R}$ are just abbreviations for $\mathscr{E}$ combined with particular coercions.)

As for (ii) above, such non-homomorphic semantic equations can easily be replaced by equivalent homomorphic ones: in the first example by "macro-expansion", in the second one by the explicit use of the fixpoint operator.

With (iii), there are two ways of getting around the non-homomorphic treatment of identifiers in binding constructs: either (a) take **Ide** as a proper semantic domain, and leave the identity semantic function $: \mathbf{Ide} \to \mathbf{Ide}$ implicit; or (b) stop considering identifiers as constituent phrases of binding constructs, taking **Ide**-indexed families of constructs instead. The two solutions are closely related: (a) leads to **Ide** arguments of semantic operations, whereas (b) leads to **Ide**-indexed families of semantic operations. We follow the ADJ group in choosing (b), as it turns out to be a useful device for dealing also with literal constants (numerals, strings, *etc.*) in semantic descriptions.

Thus we can indeed "reduce" Denotational Semantics to Initial Algebra Semantics. But we can of course go the other way, just by forgetting the algebraic structure of abstract syntax (making it into a flat domain, to be more precise). So the conclusion is that the two approaches are essentially equivalent after all.

Consistency and Completeness

When giving algebraic specifications of abstract data types, it is important to be able to check somehow that the axioms specified are the "right" ones. If a standard model for the desired abstraction already exists (*e.g.* with mathematical ADTs such as the Integers) it can be checked that the meaning of a specification is isomorphic to that model. But in other cases, there may be no *a priori* model, and then resort has to be made to other criteria.

Often, an ADT isn't specified *in vacuo*, but it is intended to be an *extension* of other ADTs. An obvious demand to make of the axioms is that they don't imply the identification of any values of the original ADTs. This is called *consistency*. Further, one can insist that the new operations don't create new values in old sorts — the axioms should be sufficient to reduce any new term of an old sort to an equivalent old term. This is the idea of *sufficient completeness* [16].

However, it seems that these notions are not so useful for ASAs, which are rather atypical as ADTs. The first problem is that ASAs have no operations giving results in standard value sorts (such as the sort **T** for truth-values): all the results of operations are "actions" (as explained in the next section). This means that the consistency condition is very weak. Secondly, it is usually the intention with a new ASA to provide operations corresponding to some new concept, which is independent of the previous concepts: this precludes making the new operations equivalent to combinations of the original ones. Finally, some ASAs introduce new sorts (indeed, whole families of them) and this situation is outside the scope of the criteria of consistency and sufficient-completeness.

How, then, can the axioms of ASAs be checked? The most promising approach seems to be to choose a set of *canonical terms* for the extended ASA, and show that the axioms do not identify canonical terms, but are sufficient to reduce arbitrary terms to canonical terms. Provided that it can be seen that these canonical terms are satisfactory representatives of "fully evaluated" actions, the axioms can be deemed to be correct.

The ASAs given below are currently being investigated regarding the completeness of their axioms with respect to a particular set of canonical terms. More axioms than the ones shown will be needed to obtain this completeness. The consistency of the axioms given is in less doubt, as a model can be given for them, based on standard domains of Denotational Semantics. (The selection of basic ASAs used in the example semantic description at the end of this paper have a simpler model than is needed for the full set, see Table 13.)

Finally, note that our use of discrete algebras, instead of continuous algebras[1] or recursion-closed algebraic theories[8], makes it difficult to achieve as good an equivalence on action terms as might be desired. With an action corresponding to a loop, for example, the axioms of the ASAs given below make the action equivalent to all its finite unfoldings. The problem is that it is only when the infinite unfolding is added to the equivalence class that one gets (for free) the equivalence with other actions representing operationally-equivalent loops that merely bring out some of the commands in the body, *e.g.* consider **repeat** $(C\ ;\ C')$ and $C\ ;$ **repeat** $(C'\ ;\ C)$: all their finite unfoldings are different, whilst their infinite unfoldings are identical. It is possible to add axioms to the specifications of the basic ASAs, reflecting such simple "program transformations", but it may turn out to be better to base ASAs on continuous algebraic foundations instead.

BASIC ABSTRACT SEMANTIC ALGEBRAS

The main idea of the proposed approach to semantics is to give semantic equations based on abstract semantic algebras that represent fundamental concepts of programming languages. This section demonstrates the feasibility of the idea by specifying some basic ASAs and illustrating their use.

These basic ASAs deal with concepts of sequence control, dataflow and scopes of bindings. They are adequate for the semantic description of most features of many present-day (imperative or functional) programming languages. Not given here are ASAs for (i) commonly-occurring data structures (arrays, files, *etc.*); (ii) "static semantics" (type-checking, *etc.*); and (iii) communicating concurrent systems. There is no difficulty in providing (i), but (ii) and (iii) are more difficult.

It is characteristic for ASAs that their elements represent so-called *actions*. These actions correspond directly to the meanings of constructs of programming languages such as commands, expressions and definitions (declarations). Common to all actions is the notion that they can be *executed* in some order (partial or total). The concept of order of execution is rather fundamental to *programming* languages, as opposed to *logic* and *specification* languages.

The operations of ASAs are either *primitive* actions (constants) or else *combinators* that give compound actions as results. Actions are not treated as operations on data, as is common in other approaches. This use of combinators for actions gives a "higher-order" nature to ASAs, yet within the standard (first-order) algebraic framework.

The conceptual analysis underlying the basic ASAs presented below is that actions may have several different *facets*, which are "orthogonal" in that they don't interfere with each other. The *imperative* facet of an action is concerned with *(side)-effects*, where the execution or not of one action influences the execution of a later one, without any explicit transfer of a value between the two actions using another facet. The *functional* facet of an action relates to dataflow: an action may *consume* and *produce* (sequences of) *values*. It is assumed that an action cannot be executed until all the values that it is to consume have been computed: this corresponds to actions being *strict* functions of their arguments. Finally an action may

Seq:

$$\textbf{(A)} \quad a \; ::= \; a_1 ; a_2 \;\mid\; () \;\mid\; \textbf{stop} \;\mid\; \textbf{error}$$

axioms

1.	$a ; (a' ; a'')$	$=$	$(a ; a') ; a''$
2.	$() ; a$	$=$	a
3.	$a ; ()$	$=$	a
4.	$\textbf{stop} ; a$	$=$	$\textbf{stop}$
5.	$\textbf{error}$	$=$	$\textbf{error} ; \textbf{stop}$

Table 1. Sequential Actions

have a *binding* facet. This facet concerns references to values *denoted* by identifiers, and the binding of values to identifiers in certain *scopes*. (There are some useful analogies between the functional and binding facets, but no mutual interference.)

Now, most of the operators of the basic ASAs below are really only interesting in connection with just one of the facets of their operands (some constant actions are used to provide interfaces between facets). The explanation of ASAs is simplified here by generally ignoring the subsidiary facets in specifications. The extension of operators to operands with subsidiary facets is discussed afterwards. (In general, the axioms of the simplified specifications are still to hold when the actions have subsidiary facets. Those that don't are marked with a dagger: †.)

Imperative Actions

In the ASA **Seq** specified in Table 1, the sort A corresponds to actions that are executed for their presumed *effect* on some notional state. For example, they might be stack-machine instructions, or database transactions. In any case, **Seq** provides an operator $a_1 ; a_2$, corresponding to the concept of the *sequential execution* of a_1 followed by a_2, and also the *empty* action that has no effect at all (usually written in parentheses thus: () to make it apparent). There is also an action **stop**, for *terminating* execution normally, *i.e.* causing the following actions in a sequence to be skipped. The action **error** is like **stop**, but corresponds to abnormal termination.

Non-Det:

$$\textbf{(A)} \quad a \; ::= \; a_1 \,\Box\, a_2 \;\mid\; a_1 , a_2 \;\mid\; < a_1 >$$

axioms

1.	$a \,\Box\, (a' \,\Box\, a'')$	$=$	$(a \,\Box\, a') \,\Box\, a''$
2.	$a \,\Box\, a'$	$=$	$a' \,\Box\, a$
3.	$a \,\Box\, a$	$=$	a
4.	$a ; (a' \,\Box\, a'')$	$=$	$(a ; a') \,\Box\, (a ; a'')$
5.	$(a \,\Box\, a') ; a''$	$=$	$(a ; a'') \,\Box\, (a' ; a'')$
6.	$a , (a' \,\Box\, a'')$	$=$	$(a , a') \,\Box\, (a , a'')$
7.	$< a \,\Box\, a' >$	$=$	$< a > \Box < a' >$
8.†	$(< a_1 > ; a_2) , (< a_1' > ; a_2')$	$=$	$(< a_1 > ; (a_2 , (< a_1' > ; a_2'))) \,\Box\, (< a_1' > ; (a_2' , (< a_1 > ; a_2)))$

Table 2. Non-Deterministic Actions

The ASA **NonDet** in Table 2 extends **Seq** with some operators for actions whose order of execution is only partially determined. The operator $a_1 \,\Box\, a_2$ gives a *non-deterministic choice* between a_1 and a_2: the execution of $a_1 \,\Box\, a_2$ entails the execution of either a_1 or of a_2, but not of both. Less general than $a_1 \,\Box\, a_2$ for expressing partial orderings on actions is a_1 , a_2, which arbitrarily *interleaves* the executions of a_1 and a_2 (the usage here of ',' and ';' is close to that in Algol68). Of course, interleaving is more than just an arbitrary choice between left-to-right and right-to-left.

The interleaving stops at the level of *indivisible* actions $< a_1 >$. (All atomic actions a introduced below should really be specified as being indivisible, by axioms $< a > = a$, but these axioms are omitted here.) When actions a and a' are indivisible and moreover *commute*: $a ; a' = a' ; a$ then there is no difference between $a ; a'$ and a , a'. But the use of the latter operator suggests that the lack of ordering on the executions of a and a' is intended (or irrelevant), and not merely incidental; whereas the use of the sequencing emphasises determinism.

When the grain of the primitive actions of **A** is fine enough so that the concurrent execution of any two primitive actions a and a' is equivalent to executing $(a ; a') \,\Box\, (a' ; a)$, then the operator $\_ , \_$ may be regarded as giving concurrent execution in general.

Note that $a_1 \, , \, a_2$ is both commutative and associative, from axiom 8. (However, axiom 8 doesn't hold in general when the actions involved have non-trivial functional facets, as then the values produced need re-ordering.)

Functional Actions

The functional facet of actions is concerned with the *data values* that they *consume* and *produce*, and with the way that data is transferred between actions during execution. Although the concept of dataflow could be represented by purely imperative actions (*e.g.* pushing and popping values on an implicit stack), it seems more appropriate to use a form of functional composition, along with tupling and projections, to express dataflow (following ADJ[3] and Backus[5]).

Actually, this author prefers to use a more elaborate ASA than the one given here for expressing dataflow (it is reminiscent of λ-notation, allowing the naming of consumed values and direct reference to them from within complex action terms). But to simplify the presentation of the proposed approach, we make do here with the more elementary ASA for tupling and projections.

Table 3 specifies a family of sorts ${}^{\sigma}\mathbf{A}^{\tau}$ of actions, indexed by their *source* $\sigma \in \Delta$ and *target* $\tau \in \Delta$. The set Δ is (for the moment) a set S^* of sequences of sorts $s \in S$ of some (language-specific) ADT **Val** of data values (usually a combination of standard ADTs like **Integer** and **Truth**). We write () for the empty sequence in S^*, just s for the unit sequence consisting of s and (σ_1, σ_2) for the (associative) concatenation of S^*. The source of an action gives the sorts of the components of the sequence of values that it consumes, the target gives those for the values that it produces. The main reason for introducing such an indexed family of sorts is to prohibit some nonsensical actions, such as passing a truth-value to an action that is expecting an integer.

The other family of sorts ${}^{\sigma}\tilde{\mathbf{A}}^{\tau}$ in Table 3 corresponds to "step-less" actions which merely deliver sequences of values that have been previously computed. (Actions in these sorts are to remain without side-effects, when the functional and imperative facets are combined below.)

Now for the concepts behind the operations specified in Table 3. $a_1 \,!\, a_2$ gives the *composition* of a_1 followed by a_2: the sequence of values produced by a_1 is consumed by a_2 (hence the execution of a_1 must precede that of a_2). $a_1 \, ; a_2$ is the same as in **Seq** when $\sigma_1 = \sigma_2 = \tau_1 = \tau_2 = ()$, and the execution of a_1 again precedes that of a_2. This is generalised here to *target tupling*, with the sequence of values consumed by the whole action being copied to both a_1 and a_2 and the sequences of values thus produced being concatenated. (Another generalisation of this is the *product* $a_1 \times a_2$ with source (σ_1, σ_2) and target (τ_1, τ_2). This is not included here in the basic ASAs since it can be derived as $(\pi_1 \,!\, a_1) \, ; (\pi_2 \,!\, a_2)$.)

The operator $\tilde{a}_1 \, , \, \tilde{a}_2$ is also target tupling, this time of step-less actions. ${}^{\sigma}()$ is the *empty* action that ignores the values sent to it and just produces the empty sequence of values. (In general,

Fun:

$({}^{\sigma}\mathbf{A}^{\tau})$	a	$::=$	$a_1 \,!\, a_2$	-- $\sigma = \sigma_1;\ \tau = \tau_2;\ \tau_1 = \sigma_2$
		$\mid$	$a_1 \, ; a_2$	-- $\sigma = \sigma_1 = \sigma_2;\ \tau = (\tau_1, \tau_2)$
		$\mid$	$\tilde{a}_1$	-- $\sigma = \sigma_1;\ \tau = \tau_1$
$({}^{\sigma}\tilde{\mathbf{A}}^{\tau})$	$\tilde{a}$	$::=$	$\tilde{a}_1 \, , \, \tilde{a}_2$	-- $\sigma = \sigma_1 = \sigma_2;\ \tau = (\tau_1, \tau_2)$
		$\mid$	${}^{\sigma}()$	-- $\tau = ()$
		$\mid$	${}^{\sigma_1, \sigma_2}\pi_i$	-- $\sigma = (\sigma_1, \sigma_2);\ \tau = \sigma_i$
		$\mid$	ι^{σ}	-- $\tau = \sigma$
		$\mid$	$@^{(s_1, \ldots, s_n)}o(\pi_1, \ldots, \pi_n)$	-- $\sigma = (s_1, \ldots, s_n);\ \tau = s;$ $o \in \Sigma\mathbf{Val}_{(s_1, \ldots, s_n), s}$

axioms

1.	$a \,!\, (a' \,!\, a'')$	$=$	$(a \,!\, a') \,!\, a''$
2.	$\iota^{\sigma} \,!\, a$	$=$	a
3.	$a \,!\, \iota^{\tau}$	$=$	a
4.	$a \, ; (a' \, ; a'')$	$=$	$(a \, ; a') \, ; a''$
5.	${}^{\sigma}() \, ; a$	$=$	a
6.	$a \, ; {}^{\sigma}()$	$=$	a
7.	$\tilde{a} \, , (\tilde{a}' \, , \tilde{a}'')$	$=$	$(\tilde{a} \, , \tilde{a}') \, , \tilde{a}''$
8.	${}^{\sigma}() \, , a$	$=$	a
9.	$a \, , {}^{\sigma}()$	$=$	a
10.†	$\tilde{a} \,!\, (a' \, ; a'')$	$=$	$(\tilde{a} \,!\, a') \, ; (\tilde{a} \,!\, a'')$
11.†	$\tilde{a} \,!\, (\tilde{a}' \, , \tilde{a}'')$	$=$	$(\tilde{a} \,!\, \tilde{a}') \, ; (\tilde{a} \,!\, \tilde{a}'')$
12.†	$(\tilde{a}_1 \, , \tilde{a}_2) \,!\, {}^{\tau_1, \tau_2}\pi_i$	$=$	$\tilde{a}_i$
13.	$({}^{\tau_1, \tau_2}\pi_1 \, , {}^{\tau_1, \tau_2}\pi_2)$	$=$	$\iota^{(\tau_1, \tau_2)}$
14.	$(\tilde{a}_1 \, , \tilde{a}_2) \,!\, (\pi_2 \, , \pi_1)$	$=$	$\tilde{a}_2 \, , \tilde{a}_1$
15.	$\tilde{a} \,!\, {}^{\tau}()$	$=$	${}^{\sigma}()$
16.	$(@o_1 \, , \ldots , @o_n) \,!\, @o(\pi_1, \ldots, \pi_n)$	$=$	$@o(o_1, \ldots, o_n)$

Table 3. Functional Actions

superscripts and even parentheses may be omitted from operators when the context makes this unambiguous.) ${}^{\sigma_1,\sigma_2}\pi_i$ with $i = 1$ or 2 *selects* the indicated subsequence of its consumed values. ι^{σ} is the identity for $a_1 \,!\, a_2$, it just passes on a sequence of values unchanged. Finally, $@^{(s_1,\ldots,s_n)}o(\pi_1,\ldots,\pi_n)$ gives a convenient way of embedding the operations of the underlying data type **Val** as (step-less) actions — mixfix notation is allowed for o.

As for the axioms, note that 10, 11 and 12 would be conceptually inappropriate with arbitrary actions a instead of step-less ones $\tilde{a}$: duplicating or removing actions is significant when they have non-null targets (or side-effects).

Coercions play an important role in ASAs, in connection with modifiability. The idea is that when there is a natural injection from one sort to another, then this is represented by an (invisible) coercion operator. In the ADT **Val** for instance, there could be a coercion from natural numbers **N** to integers **Z**. As discussed in the previous section, this allows polymorphic operators like **0** and _ + **1** to be used without it being apparent that there is in fact a hierarchy of sorts.

Apart from their use in coping with polymorphic operators, coercions between sorts are also used in ASAs for exhibiting the grouping of data into denotable values, storable values, *etc.*, in the same way that sum domains are used in standard Denotational Semantics[15]. (The usual convention of omitting injections into sum domains is followed here, but projections are always explicit, as described below.)

Coercions between the sorts of **Val** induce a partial ordering on the set S, and this is now extended componentwise to S^*. (They induce also some coercions between action sorts, corresponding to values being coerced during transfer from one action to another.) The operator for *source tupling* (the dual of target tupling) is $a_1 \,/\, a_2$, which selects a_1 or a_2 for execution depending on whether the sequence of values consumed was coerced from σ_1 or from σ_2.

Just as we needed Δ closed under concatenation of sequences (σ_1,σ_2) for target tupling and projections, the use of coercions and source tupling entails that Δ is to be closed under the corresponding sum operation, which we write as $(\sigma_1 \mid \sigma_2)$. For good measure, we close Δ also under finite sequences of arbitrary length, written $\sigma_1{}^*$ — we assume $\sigma^* = () \mid (\sigma,\sigma^*)$. Finally, when all the operators of a sort s are coercions from other sorts $s_1,\ldots,s_n$ then $(s_1 \mid \ldots \mid s_n) = s$ in Δ.

With this enriched Δ, Table 4 specifies source tupling and coercions between actions in terms of the coercions between the sorts S of **Val**.

Note that source tupling can be used to give the common conditional action, where the selection between two actions is determined by a truth value. Let **T** be the sort for truth values in **Val**, and

Sum:

$$
\begin{array}{llll}
({}^{\sigma}\mathbf{A}^{\tau}) \quad a & ::= & a_1 & \text{-- } (\sigma_1 \to \tau_1) < (\sigma \to \tau)\text{--see below} \\
 & \mid & a_1 \,/\, a_2 & \text{-- } \sigma = (\sigma_1 \mid \sigma_2);\ \tau = \tau_1 = \tau_2
\end{array}
$$

axioms

$$
\begin{array}{lrcll}
1. & \tilde{a} \,!\, (a' \,/\, a'') & = & \tilde{a} \,!\, a' & \text{-- } \tau = \sigma' \\
2. & \tilde{a} \,!\, (a' \,/\, a'') & = & \tilde{a} \,!\, a'' & \text{-- } \tau = \sigma'' \\
3. & a \,/ (a' \,/\, a'') & = & (a \,/\, a') \,/\, a'' & \\
4. & a \,/\, a' & = & a' \,/\, a &
\end{array}
$$

Single-step Coercions

The binary relations $\sigma_1 < \sigma_2$ on Δ and $(\sigma_1 \to \tau_1) < (\sigma_2 \to \tau_2)$ on $\Delta \times \Delta$ are the least ones satisfying:

$$
\begin{array}{rll}
s < s' & \textit{iff} & \_ \in \Sigma\mathbf{Val}_{(s).s'} \\
\left.\begin{array}{r} (\sigma_1 \mid \ldots \mid \sigma_n) < (\sigma_1' \mid \ldots \mid \sigma_n') \\ (\sigma_1,\ldots,\sigma_n) < (\sigma_1',\ldots,\sigma_n') \end{array}\right\} & \textit{iff} & \sigma_i < \sigma_i' \text{ for some } i, \text{ all other } \sigma_j = \sigma_j' \\
\sigma^* < \sigma'^* & \textit{iff} & \sigma < \sigma' \\
(\sigma_1 \to \tau_1) < (\sigma_2 \to \tau_2) & \textit{iff} & \sigma_2 < \sigma_1 \text{ and } \tau_1 < \tau_2
\end{array}
$$

Table 4. Coercions and Selections

let **tt** and **ff** be regarded as *sorts* rather than as constants. Taking **tt** and **ff** ambiguously as variables of the corresponding sorts, we can specify

$$
\begin{array}{llll}
(\mathbf{T}) & t & ::= & \mathbf{tt} \mid \mathbf{ff} \\
(\mathbf{tt}) & \mathbf{tt} & ::= & () \\
(\mathbf{ff}) & \mathbf{ff} & ::= & ()
\end{array}
$$

and then ${}^{\mathbf{tt}}a_1^{\tau} \,/\, {}^{\mathbf{ff}}a_2^{\tau}$ gives a conditional consuming a truth value in **T** and selecting the appropriate a_i. For selecting between actions a, a' with a null source, $\iota^{\mathbf{tt}} \,!!\, a \,/\, \iota^{\mathbf{ff}} \,!!\, a'$ can be used, the consumed truth value being thrown away by the empty action in !!.

With source tupling available, iterative actions are specified as in Table 5. Note that **while** a_1 **do** a_2 is derivable from the more fundamental **repeat** a_1.

The next basic ASA deals with the concept of functional (and procedural) abstraction, see Table 6. We now assume that the sorts of **Val** (and hence Δ) include **F** (for values representing abstracted actions) and **P** (for values that may be passed to, and returned by, abstracted

Iter:

$$\begin{array}{llll}
({}^{\sigma}\mathbf{A}^{\tau}) & a ::= & \mathbf{repeat}^{\tau}\, a_1 & \text{--}\ \ \sigma = \sigma_1;\ \tau_1 = (\sigma_1 \mid \tau) \\
 & \quad\ \mid & \mathbf{while}\, a_1\, \mathbf{do}\, a_2 & \text{--}\ \ \sigma = \sigma_1 = \sigma_2 = \tau_2 = \tau;\ \tau_1 = \mathbf{T}
\end{array}$$

axioms

$$\begin{array}{lrcl}
1. & \mathbf{repeat}^{\tau}\, a_1 & = & a_1\ !\ ((\mathbf{repeat}^{\tau}\, a_1)\ /\ \iota^{\tau}) \\
2. & \mathbf{while}\, a\, \mathbf{do}\, a' & = & (a\ ;\ \iota^{\tau'})\ !\ ({}^{\mathrm{tt},\tau'}\pi_2\ !\ a_2\ !\ \mathbf{while}\, a\, \mathbf{do}\, a')\ /\ ({}^{\mathrm{ff},\tau'}\pi_2)
\end{array}$$

Table 5. Iterative Actions

actions). Recall that @o is an action that just produces the constant value o of **Val**. So @**abstract**a_1 produces the value **abstract**a_1 in **F**, and this value may then be passed around before eventually being executed with a suitable source by the action **apply**. The operator **supply** is related to Currying in λ-notation. (Note that these are the only actions dealing with **P** and **F**.)

Before the binding facet of actions is introduced, let us consider combining the imperative and functional facets of actions — this is necessary if we are to allow operators corresponding to the concept of *storing* consumed values in *variables*, for example.

Let **Val** include sorts **L** for variables (or "locations") and **R** for storable values. Table 7 gives a primitive action **update** that has the effect of assigning its consumed storable value to its consumed variable; the action **contents** retrieves the last-assigned value of a variable. **L** is not taken to be just **Ide**, as assigning to identifiers in some programming languages can be conceptually more complex than just assigning to the variables they denote.

Abs:

$$\begin{array}{llll}
(\mathbf{F}) & f ::= & \mathbf{abstract}\, a_1 & \\
({}^{\sigma}\mathbf{A}^{\tau}) & a ::= & \mathbf{apply} & \text{--}\ \ \sigma = (\mathbf{F}, \mathbf{P}^*);\ \tau = \mathbf{P} \\
 & \quad\ \mid & \mathbf{supply} & \text{--}\ \ \sigma = (\mathbf{F}, \mathbf{P}^*);\ \tau = \mathbf{F}
\end{array}$$

axioms

$$\begin{array}{lrcl}
1. & ((@\mathbf{abstract}\, a)\ ;\ a')\ !\ \mathbf{apply} & = & a'\ !\ a \\
2. & (((@\mathbf{abstract}\, a)\ ;\ a')\ !\ \mathbf{supply})\ ;\ a'')\ !\ \mathbf{apply} & = & (a'\ ;\ a'')\ !\ a \\
3. & (((@\mathbf{abstract}\, a)\ ;\ a')\ !\ \mathbf{supply})\ ;\ a'')\ !\ \mathbf{supply} & = & \\
 & & & ((@\mathbf{abstract}\, a)\ ;\ a'\ ;\ a'')\ !\ \mathbf{supply}
\end{array}$$

Table 6. Abstracted Actions

Store:

$$\begin{array}{llll}
({}^{\sigma}\mathbf{A}^{\tau}) & a ::= & \mathbf{update} & \text{--}\ \ \sigma = (\mathbf{L}, \mathbf{R});\ \tau = () \\
 & \quad\ \mid & \mathbf{contents} & \text{--}\ \ \sigma = \mathbf{L};\ \tau = \mathbf{R} \\
 & \quad\ \mid & \mathbf{allocate} & \text{--}\ \ \sigma = ();\ \tau = \mathbf{L}
\end{array}$$

axioms

$$\begin{array}{lrcl}
1. & (\pi_1\ ,\ \pi_2)\ !\ \mathbf{update}\ ;\ \pi_1\ !\ \mathbf{contents} & = & (\pi_1\ ,\ \pi_2)\ !\ \mathbf{update}\ ;\ \pi_2 \\
2. & (!\ \mathbf{allocate}\ ;\ \pi_2)\ !\ \mathbf{update}\ ;\ \pi_1\ !\ \mathbf{contents} & = & \\
 & & & \pi_1\ !\ \mathbf{contents}\ ;\ (!\ \mathbf{allocate}\ ;\ \pi_2)\ !\ \mathbf{update} \\
3. & (\pi_1\ ,\ \pi_2)\ !\ \mathbf{update}\ ;\ (\pi_1\ ,\ \pi_3)\ !\ \mathbf{update} & = & (\pi_1\ ,\ \pi_3)\ !\ \mathbf{update} \\
4. & (!\ \mathbf{allocate}\ ;\ \pi_2)\ !\ \mathbf{update}\ ;\ (\pi_1\ ,\ \pi_3)\ !\ \mathbf{update} & = & \\
 & & & (\pi_1\ ,\ \pi_3)\ !\ \mathbf{update}\ ;\ (!\ \mathbf{allocate}\ ;\ \pi_2)\ !\ \mathbf{update}
\end{array}$$

Table 7. Storing Values in Variables

The action **allocate** produces a "fresh" variable; it must also have a side-effect, as its execution influences the variable produced by the next **allocate**. (Garbage collection is usually ignored in standard Denotational Semantics, but one might wish to introduce an action **dispose**, making a variable passed to it re-usable.)

The operators specific to imperative actions, *e.g.* **stop** and $a_1\ [\!]\ a_2$, should now be extended with the functional facet. This is quite straightforward and the details are omitted here. All the axioms so far may be taken unchanged for actions with both imperative and functional facets, except for the daggered axiom 8 of **Seq**, which needs a permuting action inserting to keep any values produced in the right order.

Binding Actions

The binding facet of actions is concerned with associations between *identifiers* and *denotable values*, and with scope rules that make bindings available in particular actions.

Table 8 specifies a family of sorts ${}^{\alpha}\mathbf{A}^{\beta}$ indexed by the sets of identifiers in **Ide** that they *access* (α) and *bind* (β). For the moment, the functional facet of actions is ignored. The operator $a_1 \div a_2$ is a *composition* for the binding facet: the bindings yielded by a_1 are made available in a_2. Moreover, these are the only bindings available in a_2, as those available outside the whole action are restricted in scope to a_1. Finally, the whole action yields only the bindings given by a_2. This rather strict scope rule gives an associative operator, in contrast to the more common scope rules to be found in most programming languages (which can be derived from the fundamental operators given here).

Scopes:

$$
\begin{array}{llcll}
({}^{\alpha}\mathbf{A}^{\beta}) & a & ::= & a_1 \div a_2 & \text{--}\ \alpha = \alpha_1;\ \beta = \beta_2;\ \beta_1 = \alpha_2 \\
 & & \mid & a_1 \,;\, a_2 & \text{--}\ \alpha = \alpha_1 = \alpha_2;\ \beta = \beta_1 \uplus \beta_2 \\
 & & \mid & \tilde{a}_1 & \text{--}\ \alpha = \alpha_1;\ \beta = \beta_1 \\
 & & \mid & a_1 & \text{--}\ \alpha = \alpha_1 \uplus \{I\};\ \beta = \beta_1 \\
 & & & & \text{--}\ \mathit{or}{:}\ \alpha = \alpha_1;\ \beta \uplus \{I\} = \beta_1 \\
({}^{\alpha}\tilde{\mathbf{A}}^{\beta}) & \tilde{a} & ::= & \tilde{a}_1 \,,\, \tilde{a}_2 & \text{--}\ \alpha = \alpha_1 = \alpha_2;\ \beta = \beta_1 \uplus \beta_2 \\
 & & \mid & \rho^{\alpha} & \text{--}\ \beta = \alpha
\end{array}
$$

axioms

$$
\begin{array}{lrcl}
1. & a \div (a' \div a'') & = & (a \div a') \div a'' \\
2. & \rho^{\alpha} \div a & = & a \\
3. & a \div \rho^{\alpha} & = & a \\[6pt]
4. & a \,;\, (a' \,;\, a'') & = & (a \,;\, a') \,;\, a'' \\
5. & \rho^{\emptyset} \,;\, a & = & a \\
6. & a \,;\, \rho^{\emptyset} & = & a \\[6pt]
7. & \tilde{a} \,,\, (\tilde{a}' \,,\, \tilde{a}'') & = & (\tilde{a} \,,\, \tilde{a}') \,,\, \tilde{a}'' \\
8. & \rho^{\emptyset} \,,\, \tilde{a} & = & \tilde{a} \\
9. & \tilde{a} \,,\, \rho^{\emptyset} & = & \tilde{a} \\[6pt]
10. & \tilde{a} \div (a' \,;\, a'') & = & (\tilde{a} \div a') \,;\, (\tilde{a} \div a'') \\
11. & \tilde{a} \div (\tilde{a}' \,,\, \tilde{a}'') & = & (\tilde{a} \div \tilde{a}') \,;\, (\tilde{a} \div \tilde{a}'') \\[6pt]
12. & \rho^{\alpha} \,,\, \rho^{\alpha'} & = & \rho^{\alpha \uplus \alpha'} \\
13.^{\dagger} & \tilde{a} \,,\, \tilde{a}' & = & \tilde{a}' \,,\, \tilde{a} \\
14. & \tilde{a} \div \rho^{\emptyset} & = & \rho^{\emptyset}
\end{array}
$$

Table 8. Scopes of Bindings

$$
\begin{array}{llcll}
({}^{\alpha;\sigma}\mathbf{A}^{\beta;\tau}) & a & ::= & a_1 \,!\, a_2 & \text{--}\ \alpha = \alpha_1 = \alpha_2;\ \beta = \beta_1 \uplus \beta_2 \\
 & & \mid & a_1 \div a_2 & \text{--}\ \sigma = \sigma_1;\ \tau = \tau_2;\ \tau_1 = \sigma_2 \\
 & & \mid & \tilde{a}_1 & \text{--}\ \sigma = \sigma_1;\ \tau = \tau_1 \\
({}^{\alpha;\sigma}\tilde{\mathbf{A}}^{\beta;\tau}) & \tilde{a} & ::= & {}^{\sigma}() & \text{--}\ \alpha = \emptyset;\ \beta = \emptyset \\
 & & \mid & {}^{\sigma_1,\sigma_2}\pi_i & \text{--}\ \alpha = \emptyset;\ \beta = \emptyset \\
 & & \mid & \iota^{\sigma} & \text{--}\ \alpha = \emptyset;\ \beta = \emptyset \\
 & & \mid & @^{(s_1,\ldots,s_n)}o(\pi_1,\ldots,\pi_n) & \text{--}\ \alpha = \emptyset;\ \beta = \emptyset \\
 & & \mid & {}^{\sigma}\rho^{\alpha} & \text{--}\ \tau = ()
\end{array}
$$

Table 9. Combination of Functional and Binding Facets

The *identity* for composition of bindings is ρ^{α}: it just passes on the bindings for all the identifiers in α. (The superscript here is usually omitted when all the available bindings are to be passed on.) Note that ρ^{α} can also be used to *project* from a larger set of bindings, thanks to the coercions between binding actions — even to give the empty binding action.

The operator $a_1 \,;\, a_2$ gives target tupling for bindings. In general the operators for the binding facet are closely related to those of the functional facet (not too surprisingly).

Now, assuming that the underlying ADT **Val** contains a sort **D** of denotable values, the action **bind**I with source **D** is just to bind the value consumed to the identifier I (in fact this is to be an **Ide**-indexed family of actions, appropriate for giving a homomorphic semantics for definitions in programming languages). The value currently bound to I is produced by the action (family) **find**I. These actions provide an interface between the functional and binding facets of actions. How should the previously-given functional operators treat bindings, and *vice versa?*

Table 9 gives some examples. The idea is to generalise each operator by allowing it to treat subsidiary facets like particular basic operators for those facets. For example, $a_1 \,!\, a_2$ behaves like $a_1 \,;\, a_2$ for bindings, and $a_1 \div a_2$ behaves like $a_1 \,!\, a_2$ on sequences of values. Some operators like $a_1 \,;\, a_2$ are basic for all the facets of actions. The axioms given separately for each facet may be taken without modification for the combined actions, except for the daggered ones in Table 3, which only hold when the duplicated (or removed) step-less actions involved yield no bindings.

Table 10 specifies **bind**I and **find**I, and illustrates how a less strict scope rule than $a_1 \div a_2$, namely $a_1 : a_2$, may be derived. This operator in fact corresponds to Algol60-like block structure, with non-local bindings being available in a_2 unless hidden by bindings of the same

Bind:

$$
\begin{array}{llcll}
({}^{\alpha:\sigma}\mathbf{A}^{\beta;\tau}) & a & ::= & a_1 : a_2 & \text{--}\ \alpha = \alpha_1 \cup (\alpha_2 - \beta_1);\ \beta = \beta_2; \\
 & & & & \quad \sigma = \sigma_1;\ \tau = \tau_2;\ \tau_1 = \sigma_2 \\
({}^{\alpha;\sigma}\tilde{\mathbf{A}}^{\beta:\tau}) & a & ::= & \mathbf{bind}\,I & \text{--}\ \alpha = \emptyset;\ \beta = \{I\};\ \sigma = \mathbf{D};\ \tau = () \\
 & & \mid & \mathbf{find}\,I & \text{--}\ \alpha = \{I\};\ \beta = \emptyset;\ \sigma = ();\ \tau = \mathbf{D} \\
 & & \mid & \tilde{a}_1\ \mathbf{!}\ \mathbf{bind}\,I & \text{--}\ \alpha = \alpha_1;\ \beta = \{I\};\ \beta_1 = \emptyset; \\
 & & & & \quad \sigma = \sigma_1;\ \tau = ();\ \tau_1 = \mathbf{D}
\end{array}
$$

axioms

$$
\begin{array}{lrcll}
1. & a : a' & = & ({}^{\sigma}\rho^{\alpha' - \beta}, a) \div a' & \\
2. & \tilde{a}\ \mathbf{!}\ \mathbf{bind}\,I \div \mathbf{find}\,I & = & \tilde{a} & \\
3. & (\tilde{a}, \tilde{a}') \div \mathbf{find}\,I & = & \tilde{a} \div \mathbf{find}\,I & \text{--}\ I \in \beta \\
4. & & = & \tilde{a}' \div \mathbf{find}\,I & \text{--}\ I \in \beta'
\end{array}
$$

Table 10. Binding Actions

identifiers in a_1. It may also be regarded as syntactic substitution; but note that it is not associative.

How about the scopes for abstractions? The aim, of course, is to be able to express both the so-called dynamic and static scope rules. Now **abstract**a_1 is a value, not an action, and the action producing this value, **@abstract**a_1, has no accessed identifiers α. Any occurrences of **find**I in a_1 are hidden for the bindings available when the abstraction is produced. They only become exposed when the abstraction is applied. This gives the dynamic scope rule for non-locally bound identifiers in abstractions.

Freeze:

$$
\begin{array}{llcll}
({}^{\alpha;\sigma}\mathbf{A}^{\beta;\tau}) & a & ::= & \mathbf{freeze}^{\alpha} & \text{--}\ \beta = \emptyset;\ \sigma = \mathbf{F};\ \tau = \mathbf{F} \\
 & & \mid & {}^{\alpha}\mathbf{apply}^{\beta} & \text{--}\ \sigma = (\mathbf{F}, \mathbf{P*});\ \tau = \mathbf{P}
\end{array}
$$

axioms

$$
\begin{array}{lrcl}
1. & \tilde{a} \div (\mathbf{@abstract}\,a)\ \mathbf{!}\ \mathbf{freeze}^{\alpha} & = & \mathbf{@abstract}((\iota, (\tilde{a} \div \rho^{\alpha})) : a) \\
2. & ((\mathbf{@abstract}\,a), \tilde{a}_1)\ \mathbf{!}\ {}^{\alpha'}\mathbf{apply}^{\beta'} & = & \tilde{a}_1 \div a \ \ \text{--}\ \ \alpha \subseteq \alpha';\ \beta' \subseteq \beta
\end{array}
$$

Table 11. Freezing Actions

However, Table 11 introduces an action **freeze**$^{\alpha}$. This action consumes a value **abstract**a_1 in **F** and produces an abstraction which is the same as the one consumed, except that all the bindings for identifiers in α have now been frozen in a_1 to be those available to the **freeze**$^{\alpha}$ action. When **freeze**$^{\alpha}$ is executed immediately after an abstraction is first produced, this gives the static scope rule (see [24] for a more leisurely discussion). Note the similarity between **freeze**$^{\alpha}$ and **supply**; and also that **apply** has indices giving its binding facet — the default values for these are $\alpha = \mathbf{Ide}$, $\beta = \emptyset$.

No more basic ASAs can be presented here, as this paper is already far too long. The major omissions here are basic ASAs for recursive bindings (which are complicated by the presence of side-effects) and exception-handling. Together, these allow a treatment of **goto** statements and labels (without modelling them on continuations).

EXAMPLE

The example in Table 12 illustrates the proposed approach of using abstract semantic algebras in semantic descriptions. The language described is M-Lisp, the syntactically-sugared applicative subset of ordinary Lisp that was used by McCarthy[19] for specifying the operational semantics of Lisp1.5.

A standard denotational description of M-Lisp has been given by Gordon[13.14]. The description given here follows Gordon's semantic analysis of M-Lisp quite closely. Readers unacquainted with M-Lisp should note that the scope rule for non-local identifiers in abstractions is the dynamic one, that there are no side-effects of evaluation, and that values passed to and from applications may only be S-expressions, not functions.

Given the basic ASAs presented in this paper, the only thing missing from the semantic description in Table 12 is the specification of an ASA called **Coercions**, providing the actions **bv**. **dv**, **fv** and **pv**. This specification is rather trivial, and can in fact be derived systematically from the coercions between the given "characteristic" sorts **B**, **D**, **F**, **P** and **E**. The corresponding actions are just identity on values coercible to the indicated sorts, and otherwise give **error**(termination).

Table 12 may perhaps not seem to be as concise as the standard domain-based denotational description of M-Lisp given by Gordon. In comparing the two descriptions, the reader is asked to take into account the unfamiliarity of the operator symbols of the ASAs here. Moreover, the use here of projections π_i and target tupling a_1 , a_2 to express dataflow does not seem (in general) to be as perspicuous as the use of bound variables in λ-notation. (As mentioned before, an ASA for naming consumed values of actions can be given.)

A standard denotational description may be derived from the one based on ASAs, by choosing an interpretation for action operators, satisfying all the axioms of their specifications, using domains. Table 13 gives an example of this. Continuations are used in modelling sequential execution purely for (λ-)notational convenience — Gordon used strict functions instead. Note that the presence of certain actions of ASAs implies a corresponding richness in interpretations, *e.g.* it is awkward to model **bind**I without the interpretation of the sort **A** being a function of some kind of environment.

Abstract Syntax				
(**Prog**)	M	::=	$F.$	
(**Func**)	F	::=	$I \mid \textbf{cons} \mid \dots \mid \lambda[[P];E] \mid \textbf{label}[I;F]$	
(**Form**)	E	::=	$I \mid S \mid F[A] \mid [C]$	
(**Pars**)	P	::=	$I;P \mid$	
(**Args**)	A	::=	$E;A \mid$	
(**Cases**)	C	::=	$E_1 \rightarrow E_2 ; C \mid$	
indices				
(**Ide**)	I	--	*standard identifiers*	
(**SExp**)	S	--	[**Lisp-Data**]	
Semantic Values				
(**E**)	e	::=	d	-- *expressible values*
(**D**)	d	::=	$b \mid f$	-- *denotable values*
(**P**)	p	::=	b	-- *passable values*
(**B**)	b	::=	S	-- *basic data*
(**T**)	t			-- *truth values*
(**F**)	f			-- *function abstractions*
$({}^{\alpha:\sigma}\mathbf{A}^{\beta;\tau})$	a			-- *actions*
indices:	$\sigma \in \Delta;\ \ \tau \in \Delta;\ \ \alpha \subseteq \textbf{Ide};\ \ \beta \subseteq \textbf{Ide}$			
auxiliary:	[**Coercions**, **Lisp-Data**]			
standard:	[**Truth**, **Seq**, **Fun**, **Sum**, **Abs**, **Scopes**, **Bind**]			

Table 12. Semantic Description of M-Lisp

How about the modifiability of the example semantic description given in Table 12? It is in fact rather easy to add new features like assignment, "fun-args" and static scopes to the described language: just reference the corresponding basic ASAs and add new semantic equations, modifying only the description of those language features directly influenced by the desired extensions. (The specification of the **Coercion** actions may need changing systematically to reflect new characteristic sorts.)

Semantic Equations

$$\begin{aligned}
\mathscr{M}:\mathbf{Prog}\to\mathbf{A} \quad &\text{--}\quad \sigma=\mathbf{P*};\ \ \tau=\mathbf{P};\ \ \alpha;\ \ \beta=\emptyset\\
\mathscr{M}[\![F.]\!] \quad &=\quad ((!\ \mathscr{F}[\![F]\!]\ !\ \mathbf{fv});\iota)\ !\ \mathbf{apply}
\end{aligned}$$

$$\begin{aligned}
\mathscr{F}:\mathbf{Func}\to\mathbf{A} \quad &\text{--}\quad \sigma=();\ \ \tau=\mathbf{E};\ \ \alpha;\ \ \beta=\emptyset\\
\mathscr{F}[\![I]\!] \quad &=\quad \mathbf{find}\,I\\
\mathscr{F}[\![\mathbf{cons}]\!] \quad &=\quad @\mathbf{abstract}((\pi_1\ !\ \mathbf{bv};\pi_2\ !\ \mathbf{bv})\ !\ @\mathbf{cons}(\pi_1,\pi_2))\\
&\ \ \ldots\\
\mathscr{F}[\![\lambda[[P];E]]\!] \quad &=\quad @\mathbf{abstract}(\mathscr{P}[\![P]\!]:\mathscr{E}[\![E]\!]\ !\ \mathbf{pv})\\
\mathscr{F}[\![\mathbf{label}[I;F]]\!] \quad &=\quad @\mathbf{abstract}((!\ \mathscr{F}[\![F]\!]\ !\ \mathbf{fv};\iota)\ !\\
&\qquad\qquad (\pi_1\ !\ \mathbf{dv}\ !\ \mathbf{bind}\,I;\iota):\mathbf{apply})
\end{aligned}$$

$$\begin{aligned}
\mathscr{E}:\mathbf{Form}\to\mathbf{A} \quad &\text{--}\quad \sigma=();\ \ \tau=\mathbf{E};\ \ \alpha;\ \ \beta=\emptyset\\
\mathscr{E}[\![I]\!] \quad &=\quad \mathbf{find}\,I\\
\mathscr{E}[\![S]\!] \quad &=\quad @S\\
\mathscr{E}[\![F[A]]\!] \quad &=\quad (\mathscr{F}[\![F]\!]\ !\ \mathbf{fv};\mathscr{A}[\![A]\!])\ !\ \mathbf{apply}\\
\mathscr{E}[\![[C]]\!] \quad &=\quad \mathscr{C}[\![C]\!]
\end{aligned}$$

$$\begin{aligned}
\mathscr{P}:\mathbf{Pars}\to\mathbf{A} \quad &\text{--}\quad \sigma=\mathbf{P*};\ \ \tau=();\ \ \alpha=\emptyset;\ \ \beta\\
\mathscr{P}[\![I;P]\!] \quad &=\quad ((^{\mathbf{P}.\mathbf{P*}}\pi_1\ !\ \mathbf{dv}\ !\ \mathbf{bind}\,I);(^{\mathbf{P}.\mathbf{P*}}\pi_2\ !\ \mathscr{P}[\![P]\!]))\,/\,(\iota^{()}\ !\ \mathbf{error})\\
\mathscr{P}[\![\ ]\!] \quad &=\quad ()
\end{aligned}$$

$$\begin{aligned}
\mathscr{A}:\mathbf{Args}\to\mathbf{A} \quad &\text{--}\quad \sigma=();\ \ \tau=\mathbf{P*};\ \ \alpha;\ \ \beta=\emptyset\\
\mathscr{A}[\![E;A]\!] \quad &=\quad (\mathscr{E}[\![E]\!]\ !\ \mathbf{pv});\mathscr{A}[\![A]\!]\\
\mathscr{A}[\![\ ]\!] \quad &=\quad ()
\end{aligned}$$

$$\begin{aligned}
\mathscr{C}:\mathbf{Cases}\to\mathbf{A} \quad &\text{--}\quad \sigma=();\ \ \tau=\mathbf{E};\ \ \alpha;\ \ \beta=\emptyset\\
\mathscr{C}[\![E_1\to E_2;C]\!] \quad &=\quad \mathscr{E}[\![E_1]\!]\ !\ \mathbf{bv}\ !\ @\mathbf{is\text{-}nil}(\pi_1)\ !\\
&\qquad (\iota^{\mathbf{tt}}\ !!\ \mathscr{C}[\![C]\!]\,/\,\iota^{\mathbf{ff}}\ !!\ \mathscr{E}[\![E_2]\!])\\
\mathscr{C}[\![\ ]\!] \quad &=\quad @\mathbf{nil}
\end{aligned}$$

Table 12, ctd.

Auxiliary

Lisp-Data:

$$\begin{array}{lll}
(\mathbf{SExp}) & S & ::=\ (S_1\,.\,S_2)\ \mid\ \mathbf{atom}\,q\ \mid\ \mathbf{cons}(S_1,S_2)\ \mid\ \mathbf{nil}\ \mid\ \ldots\\
(\mathbf{T}) & t & ::=\ \mathbf{is\text{-}nil}(S)\\
(\mathbf{Q}) & q & \text{--}\ \ \textit{standard quotations}
\end{array}$$

axioms

$$\begin{array}{lrcl}
1. & \mathbf{cons}(S,S') & = & (S\,.\,S')\\
2. & \mathbf{nil} & = & \mathbf{atom}\,\text{"NIL"}\\
3. & \mathbf{is\text{-}nil}(\mathbf{atom}\,q) & = & q\equiv\text{"NIL"}\\
4. & \mathbf{is\text{-}nil}((S\,.\,S')) & = & \mathbf{ff}
\end{array}$$

...

Table 12, ctd.

Of course, this is no proof that difficulties cannot arise when adding arbitrary new features; but it does suggest that the modifiability of semantic descriptions based on the ASAs presented here may be somewhat better than that of standard denotational descriptions based on domains.

Apart from good modifiability, a stated aim of the proposed approach is that parts of semantic descriptions be standard and re-usable. It is hopefully evident that the basic ASAs given above are not at all biased towards M-Lisp, and could be used (in different constellations) in semantic descriptions of many other programming languages.

Acknowledgments
The development during the last five years of the approach presented here has been helped along by patient criticism, questions and advice from many, including Dines Bjørner, Rod Burstall, Hartmut Ehrig and the group at TU-Berlin, Joe Goguen, Mike Gordon, Neil Jones, Robin Milner, Mogens Nielsen, Gordon Plotkin, Erik Meineche Schmidt, Dana Scott, Ravi Sethi, Joe Stoy, Bob Tennent, Jim Thatcher, Mitch Wand and Glynn Winskel. The students at Aarhus provided useful feed-back during courses in which the first attempts at expressing these ideas were made. Finally, the inspiration and education given by the late Christopher Strachey have been invaluable in this work.

Semantic Domains

$$
\begin{array}{rlcl}
e \in & \mathbf{E} & = & \mathbf{D} \\
d \in & \mathbf{D} & = & \mathbf{B} + \mathbf{F} \\
p \in & \mathbf{P} & = & \mathbf{B} \\
b \in & \mathbf{B} & = & \mathbf{SExp} \\
t \in & \mathbf{T} & \text{--} & \textit{standard truth values} \\
f \in & \mathbf{F} & = & {}^{\alpha;\mathbf{P}^*}\mathbf{A}^{\emptyset;\mathbf{P}} \\
a \in & {}^{\alpha;\sigma}\mathbf{A}^{\beta:\tau} & = & \mathbf{K}^{\beta:\tau} \rightarrow \mathbf{K}^{\alpha:\sigma} \\
\tilde{a} \in & {}^{\alpha;\sigma}\tilde{\mathbf{A}}^{\beta;\tau} & = & \mathbf{Env} \times \mathbf{V}^{\sigma} \rightarrow \mathbf{Env} \times \mathbf{V}^{\tau} \\
\kappa \in & \mathbf{K}^{\alpha;\sigma} & = & \mathbf{Env} \rightarrow \mathbf{V}^{\sigma} \rightarrow \mathbf{Ans} \\
\rho \in & \mathbf{Env} & = & \mathbf{Ide} \rightarrow \mathbf{D} \\
v \in & \mathbf{V}^{(s_1,\ldots,s_n)} & = & s_1 \times \cdots \times s_n \\
 & \mathbf{Ans} & \text{--} & \textit{unspecified}
\end{array}
$$

Operations

$$
\begin{array}{lrcl}
\mathbf{A}: & (a_1 \,!\, a_2)\kappa\rho v & = & a_1(\lambda\rho_1 v_1 . a_2(\lambda\rho_2 v_2 . \kappa(\rho_1[\rho_2])v_2)\rho v_1)\rho v \\
\mathbf{A}: & (a_1 \,;\, a_2)\kappa\rho v & = & a_1(\lambda\rho_1 v_1 . a_2(\lambda\rho_2 v_2 . \kappa(\rho_1[\rho_2])(v_1, v_2))\rho v)\rho v \\
\mathbf{A}: & (a_1 : a_2)\kappa\rho v & = & a_1(\lambda\rho_1 v_1 . a_2\kappa(\rho[\rho_1])v_1)\rho v \\
\mathbf{A}: & (a_1 \,/\, a_2)\kappa\rho v & = & v?\sigma_1 \rightarrow a_1\kappa\rho(v \mid \sigma_1),\ a_2\kappa\rho(v \mid \sigma_2) \\
\mathbf{A}: & (\mathbf{apply})\kappa\rho(f, p^*) & = & f\kappa\rho p^* \\
\mathbf{A}: & (\mathbf{error})\kappa\rho v & = & (\text{“Error”}, v) \\
\mathbf{A}: & (\tilde{a}_1)\kappa\rho v & = & (\lambda(\rho', v').\kappa\rho' v')(\tilde{a}_1(\rho, v)) \\
 & & & \\
\tilde{\mathbf{A}}: & (\mathbf{bind}\, I)(\rho, d) & = & ([I \leftarrow d], ()) \\
\tilde{\mathbf{A}}: & (\mathbf{find}\, I)(\rho, ()) & = & ([], \rho(I)) \\
\tilde{\mathbf{A}}: & ()(\rho, v) & = & ([], ()) \\
\tilde{\mathbf{A}}. & (\pi_i)(\rho, (v_1, v_2)) & = & ([], v_i) \\
\tilde{\mathbf{A}}: & (\iota)(\rho, v) & = & ([], v) \\
\tilde{\mathbf{A}}: & (@\, o(\pi_1, \ldots, \pi_n))(\rho, (v_1, \ldots, v_n)) & = & ([], o(v_1, \ldots, v_n)) \\
 & & & \\
\mathbf{F}: & (\mathbf{abstract}\, a_1) & = & a_1
\end{array}
$$

Table 13. A Model for M-Lisp

REFERENCES

1. ADJ (Goguen,J.A., Thatcher,J.W., Wagner,E.G. and Wright,J.B.), Initial algebra semantics and continuous algebras, J.ACM 24 (1977) 68-95.
2. ADJ (Goguen,J.A., Thatcher,J.W., Wagner,E.G. and Wright,J.B.), An initial algebra approach to the specification, correctness and implementation of abstract data types, in: Yeh,R.(ed.), Current Trends in Programming Methodology IV (Prentice-Hall,1979).
3. ADJ (Thatcher,J.W., Wagner,E.G. and Wright,J.B.), More on advice on structuring compilers and proving them correct, in: Proc. ICALP 1979, Graz, LNCS 71 (Springer).
4. ADJ (Ehrig,H., Kreowski,H.-J., Thatcher,J.W., Wagner,E.G. and Wright,J.B.), Parameterized data types in algebraic specification languages, in: Proc. ICALP 1980, Noordwijkerhout, LNCS 85 (Springer).
5. Backus,J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Comm.ACM 21 (1978) 613-641.
6. Broy,M. and Wirsing,M., Programming languages as abstract data types, in: Proc. 5eme Colloque de Lille, 1980.
7. Christiansen,H. and Jones,N.D., Control flow treatment in a simple semantics-directed compiler generator, to appear in: Proc. IFIP Working Conf. on Formal Description of Programming Concepts II, Garmisch, 1982 (North-Holland).
8. Gallier,J.H., Recursion-closed algebraic theories, JCSS 23 (1981) 69-105.
9. Gaudel,M.-C., Deschamp,Ph. and Mazaud,M., Semantics of procedures as an algebraic abstract data type, Rapport de Recherche No. 334, INRIA, Rocquencourt (Dec.1978).
10. Goguen,J.A., Some design principles and theory for OBJ-0, in: Proc. Int. Conf. on Math. Studies of Inf. Proc., Kyoto, Japan, 1978.
11. Goguen,J.A., Order sorted algebras, Semantics and Theory of Computation Rep. No. 14 (1978), to appear in JCSS.
12. Goguen,J.A. and Parsaye-Ghomi,K., Algebraic denotational semantics using parameterized abstract modules, in: Proc. Int. Coll. on Formalization of Programming Concepts, Peniscola, 1981, LNCS 107 (Springer).
13. Gordon,M.J.C., Models of Pure Lisp, Ph.D. Thesis, Univ. of Edinburgh (1973).
14. Gordon,M.J.C., Towards a semantic theory of dynamic binding, Computer Science Dept. Rep. No. STAN-CS-75-507, Stanford Univ. (Aug.1975).

15. Gordon,M.J.C., The Denotational Description of Programming Languages (Springer, 1979).
16. Guttag,J.V. and Horning,J.J., The algebraic specification of abstract data types, Acta Inf. 10 (1978) 27-52.
17. INRIA, Formal Definition of the ADA Programming Language, Preliminary Version, INRIA, Rocquencourt (Nov.1980).
18. Liskov,B.H. and Zilles,S.N., Specification techniques for data abstractions, IEEE SE-1 (1975) 7-18.
19. McCarthy,J. *et al.*, Lisp1.5 Programmers Manual, MIT Press (1969).
20. Milne,R.E. and Strachey,C., A Theory of Programming Language Semantics (Chapman and Hall, John Wiley, 1976).
21. Mosses,P.D., The mathematical semantics of Algol60, Tech. Mono. PRG-12, Programming Research Group, Oxford Univ. (1974).
22. Mosses,P.D., SIS — Semantics Implementation System: Reference Manual and User Guide, DAIMI MD-30, Computer Science Dept., Aarhus Univ. (Aug.1978).
23. Mosses,P.D., A constructive approach to compiler correctness, in: Proc. ICALP 1980, Noordwijkerhout, LNCS 85 (Springer).
24. Mosses,P.D., A semantic algebra for binding constructs, in: Proc. Int. Coll. on Formalization of Programming Concepts, Peniscola, 1981, LNCS 107 (Springer).
25. Pair,C., Types abstrait et semantique algebrique des langages de programmation, Rapport 80-R-011, Centre de Recherche en Informatique de Nancy (1980).
26. Raoult,J.-C. and Sethi,R., On metalanguages for a compiler-generator: properties of a notation for combining functions, to appear in: Proc. ICALP 1982, Aarhus (Springer).
27. Reynolds,J.C., Using category theory to design implicit coercions and generic operators, in: Proc. Workshop on Semantics-Directed Compiler Generation, Aarhus, 1980, LNCS 94 (Springer).
28. Rus,T., Context-free algebra: a mathematical device for compiler specification, in: Proc. MFCS 1976, Gdansk, LNCS 45 (Springer).
29. Scott,D.S., Data types as lattices, SIAM J.Comput. 5 (1976) 522-587.
30. Scott,D.S. and Strachey,C., Toward a mathematical semantics for computer languages, Tech. Mono. PRG-6, Programming Research Group, Oxford Univ. (1971).
31. Sethi,R. and Tang,A., Constructing call-by-value continuation semantics, in: Proc. ICALP 1979, Graz, LNCS 71 (Springer).
32. Strachey,C., Fundamental Concepts in Programming Languages, unpublished lecture notes (1967).
33. Tennent,R.D., The denotational semantics of programming languages, Comm.ACM 19 (1976) 437-453.
34. Wand,M., First-order identities as a defining language, Acta Inf. 14 (1980) 337-357.
35. Wand,M., Semantics-directed machine architecture, in: Proc. ACM Symp. on Principles of Programming Languages, 1982.