FIRST-ORDER IDENTITIES AS A DEFINING LANGUAGE

Mitchell Wand

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 29

FIRST-ORDER IDENTITIES AS A DEFINING LANGUAGE

MITCHELL WAND

REVISED: MAY, 1979

Research reported herein was supported in part by the National Science Foundation under Grant number MCS75-06678 A01.

Abstract: Inverting the adage that a data type is just a simple programming language, we take the position that a programming language is, semantically, just a complex data type; evaluation of a program is just another operation in the data type. The algebraic approach to data types may then be applied. We make a distinction between specification and modelling, and we emphasize the use of first-order identities as a specification language rather than as a tool for model-building. Denotational and operational semantics are discussed. Techniques are introduced for proving the equivalence of specifications. Reynolds' lambda-calculus interpreter is analyzed as an example.

A definitional interpreter constitutes a Hoare-like abstract data type in which the values consist of all the possible program phrases to be interpreted and all the possible data values which may arise during the course of a computation. The functions in the data type consist of elementary functions for manipulating the underlying data, functions for building and decomposing program phrases, and the "serious" functions which evaluate the program phrases.

The success of such an enterprise, however, depends on one's choice of a defining language. Typically, the defining language is chosen to be a subset (often proper) of the defined language.* One is then left with the task of defining the defining language, and defining the values it manipulates.

It is at this point that we interject some algebra. Instead of writing down a more-or-less "concrete" calculation in some defining language, we write a specification for the data type by writing down a series of algebraic identities which relate the various functions in the data type [16, 27]. Such a translation looks like a typical metacircular definition, but is in fact a set of axioms which may be interpreted by the well-known machinery of mathematical logic. Furthermore, by restricting ourselves to first-order identities, we can use the many results in

(*) This is called a "metacircular definition" [38].

universal algebra to answer interesting questions about such specifications.

Let us consider a simple example of how first order identities may be used as a defining language. Reynolds [38] provides a series of interpreters for a simple language with recursion. Each function in his interpreters consists of a conditional expression. Each test turns out to be a test for membership in a subtype, so the order of the tests is immaterial. The archetypical situation in the interpreter is:

```
eval = λ(r,e)...
        appl?(r) → apply(eval(opr(r),e),eval(opnd(r),e))
```

If appl?(r) is true, then r must be of the form mk-appl[x_1,x_2] where x_1 and x_2 are of type EXP; then opr(r) = x_1, opnd(r) = x_2. We deduce that for all x_1,x_2 ε EXP, e ε ENV,

eval[mk-appl[x_1,x_2],e] = apply[eval[x_1,e],eval[x_2,e]] (*)

In this formulation, the concept of "sub-type" becomes superfluous, as do the selector functions opr and opnd.

Equation (*) is nothing more than an axiom relating eval, apply, and mk-appl. Scott and Strachey [43] call such axioms "semantic equations" and use them to construct lattice-

theoretic semantics. Goguen et al. [13] interpret (*) as saying that eval is the unique homomorphism from the "initial algebra" with mk-appl as a binary operator to another algebra A with the binary operator apply, the environment parameter e being eliminated by a clever choice of carrier for the algebra A.

We adopt the following interpretation for axiom (*) : A model for axiom (*) is a set A with three binary operations together satisfying (*) . So long as we restrict ourselves to first-order identities like (*) , the theory of universal algebras gives a general construction for such models.

In this paper, we will give some examples of specifications written in this style. We will also consider the question of equivalence (or isomorphism) of specifications: when do two specifications specify the same programming language? Using these techniques, we will justify some well-known program transformations and show some relationships between different semantic models. Other questions which may be attacked by these methods are operational semantics of algebraic specifications [48] and the modular composition of specifications [4].

Section 2 gives a highly condensed summary of the mathematical machinery. Section 3 gives a more detailed overview of the algebraic approach. Section 4 gives a moderate-sized example. Section 5 gives some theorems relating Backus' RED languages [1] and Hewitt's actors [20]. In Section 6 we conclude with some remarks concerning related semantic methods.

2. Preliminaries

As indicated by the examples in the introduction, we propose to allow a variety of implementations of a semantic theory by defining an implementation of a theory to be a set A with operations Ω, satisfying some identities Δ. In terms of universal algebra, the set of implementations of the semantic theory presented by (Ω,Δ) is just the variety (or equational class) of (Ω,Δ)- algebras. Furthermore, one would like to characterize a semantic theory independent of the details of its presentation, so that different theories may be compared. Again, universal algebra supplies such an object, called an algebraic theory.

2.1 Universal Algebra

In this section, we summarize those portions of universal algebra which we will use explicity. The reader seeking a tutorial on these matters should consult [6, 13, 15, 44].

Our basic model of "a set with additional structure" is a universal algebra [6, 15]: a set A with a set of functions $A^n \to A$. One almost always deals not with a single such algebra but with a class of algebras and "structure-preserving maps" (commonly called homomorphisms) between them. In order for this notion to make sense, some correspondence must be established between the sets of functions in different algebras, or between the functions in an al-

gebra and the function names in the specification. This is done as follows:

Let ω denote the nonnegative numbers $0,1,2,\ldots$. A ranked set is a map $\Omega:S \to \omega$ for some set S . If $s \in S$, we say Ωs is the rank of s . Alternatively, if $\Omega s = n$, we say s is an n-ary member of S . When no ambiguity results, we will write Ω for S : e.g., if $s \in \Omega$, then $\Omega s \in \omega$. We will often specify Ω by its graph: e.g., $\Omega = \{(+,2),(e,1)\}$.

If Ω is a ranked set, an Ω-algebra A consists of a set, called the carrier, also denoted A , and for each $s \in \Omega$ a map $As:A^{\Omega s} \to A$

Example. Let $\Omega=\{(+,2),(e,1)\}$, $A=\omega$, $A+:\omega^2\to\omega:(x,y) \mapsto x^y$ and $Ae:\omega^0\to\omega:() \mapsto 3$.

If A and B are Ω-algebras, a morphism (or homomorphism) from A to B is a map $f:A \to B$ such that for all $s \in \Omega$, if $\Omega s = n$ and $(a_1,\ldots,a_n) \in A^n$, then

$$f(As(a_1,\ldots,a_n)) = Bs(fa_1,\ldots,fa_n)$$

If A and B are Ω-algebras, their product $A \times B$ is the Ω-algebra whose carrier is the Cartesian product of the carriers of A and B , and whose operations are performed componentwise. The projection maps $e_1:A \times B \to A$ and $e_2:A \times B \to B$ are then Ω-algebra homomorphisms.

If $\Omega:S \to \omega$ is a ranked set and X is any set (assumed disjoint from S), the set W_Ω^X of Ω-X-words is defined to be the smallest set $V \subset (S \cup X)^+$ such that

(i) $X \subset V$

(ii) if $\Omega s = 0$, then $s \in V$

(iii) if $\Omega s = n$ and $w_1,\ldots,w_n \in V$,

then $sw_1\ldots w_n \in V$

A string in W_Ω^X may be regarded as a tree in prefix Polish notation, or alternatively, as a term with function symbols from Ω and variables from X . The set W_Ω^X may be made into an Ω-algebra F_Ω^X by setting, for each $s \in S$, $F_\Omega^X s:(W_\Omega^X)^{\Omega s} \to W_\Omega^X:(w_1,\ldots,w_{\Omega s}) \mapsto sw_1\ldots w_{\Omega s}$. If A is any Ω-algebra, and f is any function $X \to A$, then f can be extended uniquely to an Ω-algebra morphism $f^*:F_\Omega^X \to A$, and every morphism $F_\Omega^X \to A$ is of this form. Since for any set A there is exactly one function $f:\phi \to A$ (the one whose graph is empty), F_Ω^ϕ is an <u>initial</u> Ω-algebra: if A is any Ω-algebra, there exists exactly one morphism $h_A:F_\Omega^\phi \to A$. If $w \in W_\Omega^X$ and $f:X \to A$ we may think of $f^*(w)$ as the element of A denoted by the word w using interpretation f for variable symbols.

It will be convenient to choose standard variables X . For $n \in \omega$ let $[n]$ denote the alphabet $\{x_1,\ldots,x_n\}$ ($[0] = \phi$) . Let $W_\Omega = \bigcup_{n\in\omega} W_\Omega^{[n]}$

<u>Example</u>. Let $\Omega=\{(+,2),(e,1)\}$. Then a typical element of $W_\Omega^{[2]}$ is $++x_1x_2x_2$. If A is as in the previous example, and $f:[2]\to\omega$ is given by $f(x_1)=2$, $f(x_2)=3$, then $f^*(++x_1x_2x_2) = (2^3)^3 = 8^3 = 512$.

An <u>n-place Ω-identity</u> is an element of $W_\Omega^{[n]} \times W_\Omega^{[n]}$. If A is an Ω-algebra and δ is an n-place Ω-identity, we say δ <u>holds</u> in A iff for every $f:[n] \to A$, $f^*(e_1(\delta)) =$

$f^*(e_2(\delta))$. If Δ is a set of Ω-identities, we say Δ holds in A iff every $\delta \in \Delta$ holds in A . A class K of Ω-algebras is an equational class (or variety) iff there exists a set Δ of identities such that $A \in K$ iff Δ holds in A . We say K is the class of (Ω,Δ)-algebras, and that (Ω,Δ) is a presentation of K . If there is any (Ω,Δ)-algebra with more than one element in its carrier, there are (Ω,Δ)-algebras of every cardinality; consequently a variety is usually a proper class (not a set). The class of (Ω,Δ)-algebras forms a category whose morphisms are the Ω-algebra homomorphisms. The category of (Ω,Δ)-algebras has an initial object.

The "algebraic approach" to data types, as presented in [16, 14] is to specify a data type by a presentation (Ω,Δ). One may distinguish several points of view concerning the significance of such a presentation. The ADJ group [14] regards (Ω,Δ) as a presentation of the initial (Ω,Δ)-algebra. We hold that the significance of the presentation is that the identities Δ tell the programmer what equalities he can expect an implementation of this data type to obey. (A similar position is taken by Guttag [17]). Thus the class of models of (Ω,Δ) is just the class of (Ω,Δ)-algebras. (For more discussion on this point, see Section 6 infra).

Now, if one adopts this latter position, one cannot deal solely with the initial algebras, since two distinct equational classes can share the same initial algebra. For example, the

one-point group is initial both in the equational class of groups and in the equational class of abelian groups. We, therefore, introduce the machinery of algebraic theories to help deal with equational classes. One may think of the presentation (Ω,Δ) as presenting the initial (Ω,Δ)-algebra and read T_Ω/Δ as denoting this algebra; this is a reasonable (initial) approximation, although this interpretation will make nonsense out of the proofs.

2.2 Algebraic Theories

Evidently, equational classes pose set-theoretical difficulties, so it is convenient to have a smaller object as a representation of a class (as ZF ordinals represent order types). There objects are categories called algebraic theories. Again the reader is referred to [30, 35] for tutorials on the concepts of category, functor, and algebraic theory.

If C is a category, $C(a,b)$ denotes the set of arrows or morphisms from object a to object b. If $f \in C(a,b)$ and $g \in C(b,c)$, their composition, a member of $C(a,c)$, is denoted $g.f$. If $f \in C(a,b)$ then $dom(f) = a$ and $cod(f) = b$. Sets will denote the category whose objects are all sets and whose arrows are all functions.

An algebraic theory is a category T whose objects are the non-negative numbers $0,1,2,\ldots$ and in which the object

n is the categorical product of the object 1 taken n times. If T is a theory, and $f_1,..,f_n \in T(k,1)$, then the product morphism in $T(k,n)$ is denoted $[f_1,..,f_n]$. We write e_i for the projection morphisms.

If Ω is a ranked set, then the free theory T_Ω may be constructed by standard methods with $T_\Omega(n,1) = W_\Omega^{[n]}$.(*) $T_\Omega(n,1)$ consists of trees whose nodes are elements of the ranked sets, whose leaves are either constant operators (i.e. whose domain is 0) or projection operators with domain n, and in which domains and codomains match appropriately throughout the tree. Composition in T_Ω is substitution of trees for leaves labelled by projection operators. $T_\Omega(n,m)$ consists of m-tuples of trees in $T_\Omega(n,1)$.

For example, let $S=\{\underline{val},\underline{alt}\}$ $\Omega=\{(\underline{val},2), (\underline{alt},3)\}$. Then

$$\underline{alt}[e_1,\underline{val}[e_1,e_2],\underline{val}[e_1,e_3]] \in T_\Omega(3,1)$$
$$[e_1,\underline{val}[e_1,e_3],\underline{val}[e_1,e_2]] \in T_\Omega(3,3)$$

and the composition of these two morphisms is

$$\underline{alt}[e_1,\underline{val}[e_1,\underline{val}[e_1,e_3]],\underline{val}[e_1,\underline{val}[e_1,e_2]]] \in T_\Omega(3,1)$$

If T_Ω is a free theory, the ranked set $\mathrm{Pairs}(\Omega)$ is

$$\{((f,f'),n) \mid f,f' \in T_\Omega(n,1)\}$$

An Ω-identity is thus an element of $\mathrm{Pairs}(\Omega)$. If $\Delta \subseteq \mathrm{Pairs}(\Omega)$ we will often write $(f,f') \in \Delta$ for $((f,f'),n) \in \Delta$. A <u>theory-functor</u> is a product-preserving functor between theories.

(*) In T_Ω, the projection morphisms $e_i \in T_\Omega(n,1)$ are just the trees $x_i \in W_\Omega^{[n]}$. We will therefore use e_i and x_i interchangeably.

If Δ is a set of identities, we construct a formal system E_Δ, whose formal objects are pairs (f,f') such that $f,f' \in T_\Omega(n,m)$ for some n and m. We denote this formal object $(f,f'):n \to m$ in order to make n and m explicit. The system E_Δ is defined as follows:

Axioms: If $((f,f'),n) \in \Delta$, then $\vdash (f,f'):n \to 1$ $E\Delta$

For any $f \in T_\Omega(n,m)$, $\vdash (f,f):n \to m$ ER

Rules: $$\frac{(f,g):n \to m}{(g,f):n \to m}\ ES \qquad \frac{(f,g):n \to m \quad (g,h):n \to m}{(f,h):n \to m}\ ET$$

$$\frac{(g,g):m \to k \quad (f,f'):n \to m \quad (h,h):p \to n}{(g.f.h, g.f'.h):p \to k}\ EC$$

$$\frac{(f_1,f'_1):m \to 1, \ldots, (f_n,f'_n):m \to 1}{([f_1,..,f_n],[f'_1,..,f'_n]):m \to n}\ EP$$

A theory may be presented by (Ω,Δ), where Ω is a ranked set (of generators) and Δ is a set of Ω-identities. (Δ,Ω) presents the theory T where $T(n,m) = T_\Omega(n,m)/E_\Delta(n,m)$, where $E_\Delta(n,m) = \{(f,f') \mid (f,f'):n \to m$ is a theorem of $E_\Delta\}$. It can be shown that the set $T(n,1)$ is the carrier of the free (Ω,Δ)-algebra on n generators. One may therefore think of the algebraic theory as representing an equational class K by all of the finitely generated free algebras in K. This generalizes the initial algebra representation, which attempts to represent the class by its free algebra on no generators.

It is easily confirmed that E_Δ preserves composition and products, that T is a theory, and that the functor $F:T_\Omega \to T$ sending each morphism to its equivalence class is a full theory functor. T is often denoted T_Ω/Δ .

If T is a theory, a T-algebra is a product-preserving functor $A:T \to Sets$. The underlying set of the algebra is $A(1)$ (we often write A for $A(1)$) . To each morphism $f \in T(n,m)$ there is a map $Af:A(n) \to A(m)$; since A is product-preserving, $Af:A^n \to A^m$. The T-algebras and natural transformations between them form a category T-Alg. The major result about algebraic theories is:

Theorem 2.1 The category of (Ω,Δ)-algebras is isomorphic to the category of T_Ω/Δ-algebras. ∎

Corollary 2.1 Let (Ω,Δ) and (Ω',Δ') be two presentations. If T_Ω/Δ and $T_{\Omega'}/\Delta'$ are isomorphic theories, than the category of (Ω,Δ)-algebras and (Ω',Δ')-algebras are isomorphic. ∎

Thus, to show that two presentations define the same class of algebras, we need only show that they present the same theory. This is often simpler than a direct proof. Another interpretation of Corollary 2.1 is that distinct equational classes always have distinct theories: it is this property that initial algebras lack, as noted above.

Corollary 2.2 Let $i:T \to T'$ be a theory functor. Then there is a forgetful functor from T'-Alg to T-Alg.

Proof. Let $i:T \to T'$ be the theory functor
Then the forgetful functor sends $A:T' \to$ Sets to $A \circ i:T \to$ Sets. ∎

For example, the theory of monoids is a subtheory of the theory of groups. Hence to every group there is a underlying monoid. See [35, pp.148-149] for generalizations and numerous examples of this result.

3. The Method

A programming language is to be modelled by a Hoare-like abstract data type in which the intended values consist of all possible program phrases and data values, and in which the operations include functions for manipulating program phrases and data values and functions for evaluating program phrases. The conventional Scott-Strachey approach adopts this view; following [21], one might prove the correctness of a implementation of a language using an "abstraction map" A whose target was an appropriate lattice.

It seems desirable, however, to maintain a stronger separation between "specification" and "implementation" or "model" [32]. It was Parnas [36] who first showed that one could write a specification of a data type (or module) independently of any implementation by concentrating on the interactions between the operations on the data type.(*) By simply refusing to write down a set of "values", this approach distinguishes itself from methods which merely provide a "model" or mathematical implementation as a standard. This approach has proved very powerful for practical problems [39, 40].

The algebraic method writes a specification for a data type by writing down a set of identities for the operations of the data type. Thus "specification" is identified with "presentation". From a presentation, we get a theory which

(*) cf: "Category theory asks of every type of Mathematical object: What are the morphisms?" [30, p.30]

is a representative of the class of all models of the specification, that is, the class of all algebras of the theory. For the purposes of this paper we identify "implementation" with "model" with "algebra".(*) Although some restrictions (e.g. nondegeneracy) might be placed on this identification, such restrictions seem to depend only on the theory and not on the specification [48].

The theory immediately gives a denotational semantics for the language: the initial algebra of the theory. Every phrase in the language has a unique meaning in the initial algebra, and these meanings satisfy the identities listed in the specifications. Furthermore, since a great deal is known about equational classes of algebras, we can use this body of knowledge as the need arises. This denotational semantics is derived automatically from the theory without the imposition of delicate a priori choices of domains, etc. [32].

The algebraic method gives not only a denotational semantics, but also an operational semantics. Operational semantics, being computational in nature, is dependent on how quantities are represented. Hence the operational semantics is presentation-dependent. Given a presentation (Ω,Δ), we construct an operational semantics for (Ω,Δ) as

(*) In the remainder of the paper, we shall use the term "model" rather than the more value-laden "implementation".

a tree-rewriting system on $T_\Omega(0,1)$ as follows [48]:

Define the set of M_Δ of Δ-moves on $T_\Omega(0,1)$ as the set of all pairs (f.g.h, f.g'.h), where $(g,g') \in \Delta$.

Proposition 3.1 [48]. Let (Ω,Δ) be a presentation such that M_Δ has the Church-Rosser property, and let $t \in T_\Omega(0,1)$. Then t is equivalent under E_Δ to some M_Δ-normal morphism t' iff (t,t') belongs to the reflexive, transitive closure of M_Δ. ∎

Thus a morphism in $T_\Omega(0,1)$, (that is to say, a tree) is transformed into a normal form by successive rewriting via the moves M_Δ. The proposition asserts that under the easily satsified Church-Rosser condition [29], this intuitive operational semantics is consistent and complete (at least for terminating computations) to the denotational semantics given by the identities. We shall see some examples of this in Section 4. (For generalizations of this proposition, see [48]; for some well-illustrated examples, [17]; and for more on tree rewriting systems, [33].)

In Section 5 we consider questions of the form: "When do two specifications specify the same class of models?" Such questions are complicated by the fact that one often deals with specifications using different sets of symbols. Corollaries 2.1 and 2.2 give a technique for attacking this problem: If two specifications yield isomorphic theories, then the model classes are isomorphic (that is, they are as close to equality as one might reasonably desire). If

(b) Let $\Delta' = \Delta - \{(u,u'.t')\} \cup \{(u,u'.t)\}$. In E_Δ , we have

$$\dfrac{\dfrac{(t,t')}{(u'.t,u'.t')}\ \text{EC}\quad \dfrac{(u,u'.t')}{(u'.t',u)}\ \text{ES}}{\dfrac{(u'.t,u)}{(u,u'.t)}\ \text{ES}}\ \text{ET}$$

So $\Delta' \subseteq E_\Delta$ and $E_{\Delta'} \subseteq E_\Delta$. Conversely, in $E_{\Delta'}$ we have

$$\dfrac{(u,u'.t)\qquad \dfrac{(t,t')}{(u'.t,u'.t')}\ \text{EC}}{(u,u'.t')}\ \text{ET}$$

So $\Delta \subseteq E_{\Delta'}$ and $E_\Delta \subseteq E_{\Delta'}$. So $E_\Delta = E_{\Delta'}$.

(c) Let $\Omega' = \Omega \cup \{s\}$ and $\Delta' = \Delta \cup \{(s,t)\}$. Construct theory functors $F: T_\Omega/\Delta \to T_{\Omega'}/\Delta'$ and $G: T_{\Omega'}/\Delta' \to T_\Omega/\Delta$ as follows: for $v \in \Omega$, let $Fv = v$ and $Gv = v$, and let $Gs = Gt$. Since $\Delta \subseteq \Delta'$, F is well-defined on T_Ω/Δ . Similarly G respects every identity in Δ' . Hence F and G are well-defined theory-functors. For $v \in \Omega$, $FGv = v$ and $GFv = v$, so G and F are inverses on terms in W_Ω . Then $FGs = FGt = t = s$ (in T_Ω/Δ). So G and F are inverses, and T_Ω/Δ is isomorphic to T_Ω'/Δ' .■

4. Examples of Algebra Semantics

4.1 Reynolds' Applicative Language

Since we started out with a single line from Reynolds [38], it is appropriate to complete the example.

Convention. Let ID denote a set of identifiers. Upper-case bold-face symbols (**N**, **SUCC**,...) will be used for particular identifiers. Upper-case italic symbols (I,I',...) will be used as metavariables ranging over identifiers (e.g. for every identifier I,...). Lower-case boldface symbols (**eval**, **cond**,...) will be used for particular elements of Ω other than identifiers. Lower-case italic symbols (opr,opnd, env,...) will be used in place of $e_1,..,e_n$ as projections.

The syntax of the defined language is given in [38] by "abstract syntax" which amounts to the following:

Definition. Let ID be a set of identifiers, and let Ω be given by:

identifier	:	1 → 1
const	:	1 → 1
appl	:	2 → 1
lambda	:	2 → 1
cond	:	3 → 1
letrec	:	3 → 1

An expression is a member of $W_\Omega^{[0]}$.

The symbols in Ω act like the constructor functions of [38]. We will see that the selectors and classifiers are

unnecessary. Similarly unnecessary are the subtypes of expressions. In [38], the first argument to lambda or letrec must be an identifier, and the second argument to letrec must be of the form lambda. We prefer to allow these arguments to be unrestricted, causing a run-time error only when they become crucial. The compile-time restriction could be simulated using many-typed theories [10,13] at the expense of a yet more rigorous propaedeutic in Section 2. The relation between single- and many-typed theories will be discussed in Section 6.

Definition. Let $\Omega' = \Omega \cup \{\text{eval}:2 \to 1, \text{initenv}:0 \to 1\}$.

An expression D is executed by starting a computation with eval.[D,initenv] .

We now begin analyzing Reynolds' Interpreter I, shown in Table 4.1. Our motto is taken from Dijkstra: "Programming [is]the art of the judicious postponement of decisions."(*) Thus any line of the interpreter which we do not know how to interpret as an identity we will defer; if we evidently need a defining language feature we will introduce a symbol for the feature. In both cases we will introduce refinement axioms when we figure out what additional properties are needed.

No axioms are needed for branches I.2 or I.3. The proper disposition of constants will be known only when we specify the elementary operations on them. Similarly,

---

(*)"Notes on Structured Programming." In Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. Structured Programming. Academic Press; London, 1972, p. 67.

although I.3 is cast in terms of a functional application, we know that an environment is less general than a function (in particular, it expects identifiers as arguments) so we need not immediately deal with the problem of higher-order quantities. So again we wait until we learn more about the nature of environments.

Line I.4, however, forces us to meet the question of functional application. Now, in the category of sets, the set Y^X of functions $X \to Y$ has associated a function $apply: Y^X \times X \to Y$ given by $apply(f,x) = f(x)$. Any set S which has the same cardinality as Y^X can serve as $X \to Y$ by using the correct application function $S \times X \to Y$. Hence we introduce a new symbol apply:2 → 1 and write

```
eval.[appl.[opr,opnd],env] = apply.[eval.[opr,env],
                                    eval.[opnd,env]]        (4.1.1)
```

Again, we will refine apply as more becomes known about the requirements on it.

Line I.5, calling evlambda, is likewise left free.

Lines I.6-7 are straightforward:

```
eval.[cond.[p,x,y],env] = choose.[eval.[p,env],
                                  eval.[x,env],             (4.1.2)
                                  eval.[y,env]]
```

which surface when we try to evaluate an identifier in line I.3. By unwinding Reynolds' interpreter, we discover:

$$\underline{\underline{eval}}.[\underline{\underline{identifier}}.probe,\underline{\underline{ext}}.[vble,value,env]] = \underline{\underline{choose}}.[\underline{\underline{eq\text{-}id}}[probe,vble],value,\underline{\underline{eval}}.[\underline{\underline{identifier}}.probe,env]] \quad (4.1.7)$$

$$\underline{\underline{eval}}.[\underline{\underline{identifier}}.probe,\underline{\underline{extrec}}.[vble,value,env]] = \underline{\underline{choose}}.[\underline{\underline{eq\text{-}id}}.[probe,vble], \underline{\underline{eval}}.[value,\underline{\underline{extrec}}.[vble,value,env]], \underline{\underline{eval}}.[\underline{\underline{identifier}}.probe,env]] \quad (4.1.8)$$

For every identifier I,

$$\underline{\underline{eq\text{-}id}}.[I,I] = \underline{\underline{true}} \quad (S4.1.9)$$

For every pair of distinct identifiers I,I',

$$\underline{\underline{eq\text{-}id}}.[I,I'] = \underline{\underline{false}} \quad (S4.1.10)$$

All that remains is to specify constants and the actions of identifiers in the initial environment. We introduce $\underline{\underline{n}}_k : 0 \to 1$ for each $k \in \omega$ as numerals. Then we have

```
eval = λ(r,e).                                                    I.1
    (const?(r) → evcon(r),                                        I.2
    var?(r) → e(r),                                               I.3
    appl?(r) → (eval(opr(r), e))(eval(opnd(r),e)),                I.4
    lambda?(r) → evlambda(r, e),                                  I.5
    cond?(r) → if eval(prem(r), e)                                I.6
        then eval(conc(r), e) else eval(altr(r),e),               I.7
    letrec?(r) → letrec e' =                                      I.8
        λx. if x = dvar(r) then evlambda(dexp(r), e') else e(x)   I.9
        ------------------------------------------------------
        in eval(body(r), e'))                                     I.10
evlambda = λ(ℓ,e).  λa. eval(body(ℓ), ext(fp(ℓ), a, e))           I.11
                    ---------------------------------
ext = λ(z, a, e). λx. if x = z then a else e(x)                   I.12
                  --------------------------
```

Table 4.1 Reynolds' Interpreter I

$$\underline{\underline{apply}}.[\underline{\underline{eval}}.[\underline{\underline{identifier}}.\underline{\underline{SUCC}},\underline{\underline{initenv}}],\underline{\underline{n}}_k] = \underline{\underline{n}}_{k+1} \qquad \text{(S4.1.11)}$$

$$\underline{\underline{apply}}.[\underline{\underline{apply}}.[\underline{\underline{eval}}.[\underline{\underline{identifier}}.\underline{\underline{EQUAL}},\underline{\underline{initenv}}],\underline{\underline{n}}_k],\underline{\underline{n}}_j]$$

$$= \begin{cases} \underline{\underline{true}} & \text{if } k = j \qquad \text{(S4.1.12)} \\ \underline{\underline{false}} & \text{if } k \neq j \qquad \text{(S4.1.13)} \end{cases}$$

Following[38], we leave the constants unspecified. One choice would be to introduce constants $\underline{\underline{c}}_k : 0 \to 1$ for $k \in \omega$ and let

$\underline{\underline{eval}}.[\underline{\underline{const}}.\underline{\underline{c}}_k,\text{env}] = \underline{\underline{n}}_k$ for every $k \in \omega$.

4.2 Another set of axioms

If the preceding axioms seem reminiscent of Reynolds' Interpreter II, we may introduce the abbreviations

get: 2 → 1

interpret: 1 → 1

evcon: 1 → 1

closure: 3 → 1

succ: 0 → 1

successor: 1 → 1

eq1: 0 → 1

eq2: 1 → 1

equal: 2 → 1

via:

get.[vble,env] = eval.[identifier.vble,env]

interpret.[exp] = eval.[exp,initenv]

evcon. c = eval.[const.c,env]

closure.[vble,body,env] = eval.[lambda.[identifier.vble,body],env]

succ = eval.[identifier.SUCC,initenv]

successor.a = apply.[succ,a]

eq1 = eval.[identifier.EQUAL,initenv]

eq2.a = apply.[eq1,a]

equal.[a,b] = apply.[eq2.a,b]

We may do some folding and unfolding to obtain the following set of axioms, which is the analogue of Interpreter II and, by Theorem 3.1, presents the same theory as that of Example 4.1:

interpret.exp = eval.[exp,initenv] 4.2.1

eval.[const.c,env] = evcon.c 4.2.2

```
eval.[identifier.vble,env] = get.[vble,env]                    4.2.3

eval.[appl.[opr,opnd],env] = apply.[eval.[opr,env],
                                    eval.[opnd,env]]           4.2.4

eval.[lambda.[identifier.vble,body],env] = closure.[vble,body,env]
                                                               4.2.5

eval.[cond.[p,x,y],env] = choose.[eval.[p,env],
                                  eval.[x,env],
                                  eval.[y,env]]                4.2.6

eval.[letrec.[vble,value,body],env] =
                  eval.[body,extrec.[vble,value,env]]          4.2.7

apply.[closure.[vble,body,env],a] =
                  eval.[body,ext.[vble,a,env]]                 4.2.8

apply.[succ,a] = successor.a                                   4.2.9

apply.[eq1,a] = eq2.a                                          4.2.10

apply.[eq2.a,b] = equal.[a,b]                                  4.2.11

get.[SUCC,initenv] = succ                                      4.2.12

get.[EQUAL,initenv] = eq1                                      4.2.13

get [probe,ext.[vble,value,env]] = choose.[eq-id.[probe,vble],
                                           value,
                                           get.[probe,env]]
                                                               4.2.14

get.[probe,extrec.[vble,value,env]] =
                  choose [eq-id.[probe,vble],
                          eval.[value,extrec [vble,value,env]],
                          get.[probe,env]]                     4.2.15

choose.[true,x,y] = x                                          4.2.16

choose.[false,x,y] = y                                         4.2.17
```

For every identifier I

$$\underline{\text{eq-id}}.[I,I] = \underline{\text{true}} \qquad \text{S4.2.18}$$

For every pair I,I' of distinct identifiers

$$\underline{\text{eq-id}}.[I,I'] = \underline{\text{false}} \qquad \text{S4.2.19}$$

For every $k \in \omega$

$$\underline{\text{eval}}.[\underline{\text{const}}.\underline{c}_k,\text{env}] = \underline{n}_k \qquad \text{S4.2.20}$$

$$\underline{\text{successor}}.\underline{n}_k = \underline{n}_{k+1} \qquad \text{S4.2.21}$$

For every $j,k \in \omega$

$$\underline{\text{equal}}.[\underline{n}_j,\underline{n}_k] = \begin{cases} \underline{\text{true}} & \text{if } j = k \qquad \text{S4.2.22} \\ \underline{\text{false}} & \text{if } j \neq k \qquad \text{S4.2.23} \end{cases}$$

This differs from Reynolds' Interpreter II in three specifics. First, it includes specification of the relevant features of the defining language. Second, recursive environment extensions created by a letrec need not save the body of the letrec-expression. (This may be deduced from the code of get, which mentions only dvar(letx(e)) and dexp(letx(e)) , but is not reflected in the abstract syntax for REC). Third, arbitrary declaring subexpressions are allowed. The restriction to lambda-expressions might be accomplished by changing 4.2.7 to

```
eval.[letrec.[name,lambda.[var,pbody],body],env]
   = eval.[body,extrecℓ.[name, var,pbody,env]]
```

and 4.2.15 to

```
get.[probe,extrecℓ.[name,var,pbody,env]] =
    choose.[eq-id.[probe,name],
            closure.[var,pbody,extrecℓ.[name,var,pbody,env]]
            get.[probe,env]]
```

4.3 Operational Semantics

We show a brief example of the operational semantics of Section 3 applied to the axioms of Section 4.2. Here is how the evaluation of ((λXX) 3) proceeds:

interpret.[appl.[lambda.[identifier.X,identifier.X], (1)
const.c_3]]

= eval.[appl.[lambda.[identifier.X,identifier.X], (2)
const.c_3],initenv]

= apply.[eval.[lambda.[identifier.X,identifier.X],initenv],
eval.[const.c_3,initenv]]

= apply.[closure.[X,identifier.X,initenv], (4)
eval.[const.c_3,initenv]]

= apply.[closure.[X,identifier.X,initenv],n_3] (5)

= eval.[identifier.X,ext.[X,n_3,initenv]] (6)

= get.[X,ext.[X,n_3,initenv]] (7)

= choose.[eq-id.[X,X],n_3,get.[X,initenv]] (8)

= choose.[true,n_3,get.[X,initenv]] (9)

= n_3 (10)

Most of the steps were merely instances of the axioms. From (3) to (4), the first occurrence of eval is rewritten, and from (4) to (5), the remaining instance of eval is similarly rewritten. In both cases, the rewrite site is in the interior of the expression. The step from (3) to

(4), for instance, is the move (f.g.h.,f.g'.h) where

f = apply.[e_1,eval.[const.c_3,initenv]

g = eval.[lambda.[identifier.e_1,e_2],e_3]

g'= closure.[e_1,e_2,e_3]

h = [X,identifier.X,initenv]

Here (g,g') is just axiom 4.2.5 (Recall that in the earlier statement of 4.2.5, vble, body, and env were just mnemonics for e_1, e_2, and e_3).

What if we had written λXZ instead of λXX? Then at steps (9) and (10), we would have gotten get.[Z,initenv], to which no moves are applicable in our operational semantics. This value may be read "the value of Z in initenv, whatever that means." But all is not lost. Under many circumstances, this is a useful property. In the presentation of Section 4.1, the expression

eval.[identifier.SUCC,initenv]

"the value of SUCC in initenv, whatever that means", plays a crucial role. Though by itself it might be taken for an error value, when given as the first argument to apply it acts as the successor function (See S4.1.11). Apply thus is capable of recovering from this "error".

Such error terms are the rule, not the exception: whenever something is required which is not immediately understood in terms of previous analysis, we leave it free and then rely on later functions to recover from the error. This happened on numerous occasions in Section 4.1. In

Section 4.2 we introduced abbreviations for several such constructs. One which persisted was extrec.[vble,value,env], which is read "the environment obtained by extending env by binding vble to value recursively, whatever that means." It became the job of get to successfully recover information from such an "error".

This feature allows a flexible treatment of errors [12,14]. To invoke a LISP analogy: our ERRSETs do not return **nil** on an error; they return the erroneous expression itself. Thus one can drive an entire system on errors, as does QLISP [37].

5. Models of Application

As an example of the use of semantic theories, we examine the "complete applicative language" model of Backus [1], and the actor model of Hewitt [20]. We will show that the semantic theory consisting of "interesting" programs in the Backus Model is just a free theory; consequently there are no nontrivial computations in a complete applicative language except those introduced by clever primitives.

Let Γ be a ranked set. We think of $\gamma \in \Gamma$ as a pattern into which substitutions are made. A typical n-ary pattern might be "form a vector of n elements". Backus calls this a "constructor syntax". Let $\Omega = \Gamma \cup \{(ap,2)\} \cup \{(\mu,1)\}$ be the ranked set obtained by adjoining to Γ a special 2-ary symbol ap (for application) and a 1-ary symbol μ (for evaluation).

For each $k \in \omega$, let $\mu_k = [\mu.e_1,..,\mu.e_k] \in T_\Omega(k,k)$.

Definition. Let B_Γ be the theory generated by Ω under the following axioms:

B1) for each $\gamma \in \Gamma$, $\mu.\gamma = \gamma.\mu_{\Gamma\gamma}$

B2) $\mu.ap = \mu.ap.\mu_2$

B3) $\mu.\mu = \mu$

Those axioms are due to Backus [1]. B1 says patterns are evaluated componentwise; B2 says the value of an

application depends only on the values of its components; and B3 says evaluation is idempotent. We will fix Γ and write B for B_Γ.

Lemma 5.1 $\mu_k.\mu_k = \mu_k$

Proof $\mu_k.\mu_k = [\mu.e_1,..,\mu.e_k].[\mu.e_1,..,\mu.e_k]$

$= [\mu.\mu.e_1,..,\mu.\mu.e_k]$

$= [\mu.e_1,..,\mu.e_k]$ ∎

Lemma 5.2 If $\Gamma\gamma = k$, $\mu.\gamma = \mu.\gamma.\mu_k$

Proof: $\mu.\gamma = \mu.\mu.\gamma = \mu.\gamma.\mu_k$ ∎

Lemma 5.3 If $t \in B(n,k)$ then $\mu_k.t = \mu_k.t.\mu_n$

Proof: By induction on the construction of Ω-words ∎

Of the morphisms in B, the interesting ones are those which are the result of an evaluation:

Definition. Let $\mu B(n,m) \subseteq B(n,m)$ be given by $\mu B(n,m) = \{\mu_m.t \mid t \in B(n,m)\}$.

Lemma 5.5 μB, with composition inherited from B, forms a category.

Proof: If $\mu_k.t \in \mu B(n,k)$ and $\mu_n.s \in \mu B(m,n)$, then $\mu_k.t.\mu_n.s = \mu_k.t.s \in \mu B(m,k)$. Hence μB is closed under composition. Associativity is inherited, and we claim that $\mu_k \in \mu B(k,k)$ is an identity arrow. $\mu_k = \mu_k.1$, so $\mu_k \in \mu B(k,k)$; to verify that μ_k is a left and right identity, we note $\mu_k.(\mu_k.t) = \mu_k.t$ and $(\mu_n.t).\mu_k = \mu_n.t$ ∎

Lemma 5.6 μB is an algebraic theory.

Proof: $\mu.\mathbf{e}_i \in \mu B(n,1)$ is the i-th projection function: $\mu.\mathbf{e}_i.(\mu.t_1,\ldots,\mu.t_n) = \mu.\mu.t_i = \mu.t_i$ ∎

Alternatively, we may think of μ as a "bug" which "activates" a node; axioms B1 and B2 cause bugs to propagate downward in the tree; axiom B3 then prevents accumulation of bugs. If $t \in B(n,1)$ then $\mu.t$ is equivalent to a tree in which every node has exactly one bug attached to it. We state this formally as follows:

Definition. Let $W'_n \subseteq W_\Omega^{[n]}$ be defined inductively by

(i) if $1 \le i \le n$, then $\mu e_i \in W'_n$

(ii) if $s \in \Omega$ and $s \neq \mu$ and $w_1,\ldots,w_{\Omega\sigma} \in W'_n$, then $\mu s w_1 \ldots w_{\Omega s} \in W'_n$

(iii) nothing else.

W'_n is the subset of trees in $W_\Omega^{[n]}$ with exactly one μ above each node.

Lemma 5.7 If $t \in W_\Omega^{[n]}$, then there exists a unique $t' \in W'_n$ such that $(\mu t, t')$ is a theorem of E_Δ.

Proof: By induction on the construction of t : if $t = \mathbf{e}_i$, then $\mu e_i \in W'_n$. If $t = \sigma w_1 \ldots w_p$, then $E_\Delta \vdash (\mu t, \mu\sigma\mu w_1 \ldots \mu w_p)$ by axiom B2 and Lemma 5.1. By the induction hypothesis, there exist $t'_1,\ldots,t'_p \in W'_n$ such that $E_\Delta \vdash (w_i, t'_i)$. Hence $(\mu t, \mu\sigma t'_1 \ldots t'_p)$ is also a theorem of E_Δ.

For uniqueness, let $H:(\Omega \cup [n])^* \to (\Omega \cup [n])^*$ given by $h(\mu) = \Lambda$ (the empty string), $h(s) = s$ $(s \in \Omega - \{\mu\})$; $h(\mathbf{e}_i) = e_i$ for $e_i \in [n]$. Then h is injective when restricted to

W'_n , and if $(w_1,w_2) \in E_\Delta$, then $h(w_1) = h(w_2)$ by an easy induction on the construction of E_Δ . ∎

Lemma 5.8 The map $W'_n \to \mu B(n,1)$ given by $t' \mapsto [t']$ mod E_Δ is a bijection ∎

Let F'_Γ be the free theory generated by $\Gamma \cup \{(\underline{ap},2)\}$.

Theorem 5.1 F'_Γ and μB_Γ are isomorphic algebraic theories.

Proof: Let $f_n : W^{[n]}_{\Gamma'} \to W^{[n]}_{\Omega}$ be given by

$f_n(x_i) = \mu x_i$

$f_0(s) = \mu s \quad \Gamma s = 0$

$f_n(sw_1 \ldots w_m) = \mu s \cdot f(w_1) \cdot \cdots \cdot f(w_m) \quad \Gamma s = m$ (where $\cdot$ denotes concatenation of strings).

f_n is evidently a bijection $W^{[n]}_{\Gamma'} \to W'_n$, so by Lemma 5.8 it extends to a bijection $W^{[n]}_{\Gamma'} \to \mu B_\Gamma(n,1)$, and extends componentwise to a family of bijections $f_{nm} : F'_\Gamma(n,m) \to \mu B_\Gamma(n,m)$. A routine calculation shows the f_{nm} are functorial. Furthermore, $x_i \in W^{[n]}_{\Gamma'}$ is sent to $[\mu x_i] = \mu . x_i \in \mu B(n,1)$, so the functor is product-preserving and hence an isomorphism of theories. ∎

The content of Theorem 5.1 is that once a computation gets started, the μ's rapidly get distributed to all the nodes in the tree, so one can equally well assume that the μ's are always present and therefore ignore them. We next discuss actor theories, which are theories of typeless application in the style of the lambda-calculus. We call these theories "actor theories" because they seem to be

the semantic theories corresponding to Hewitt's actors [20].

Definition. Let X be any set. The free theory of actors with primitive actors X is the free theory generated by $\{(x,0)|x\varepsilon X\} \cup \{(\underline{send},2)\}$, and is denoted A_X.

An A_X-algebra C is a set (of "actors"), with distinguished elements Cx for each $x \varepsilon X$ and a binary operation $C\underline{send}$ (transmission) such that $C\underline{send}(a,b)$ is the actor which results from sending the message b to target a. We write $\langle a\ b\rangle$ for $\underline{send}(a,b)$ and we make the convention that transmission associates to the left; thus $\langle a_1 \ldots a_n\rangle = \langle\langle\ldots\langle a_1\ a_2\rangle\ a_3\rangle\ldots a_n\rangle$.

Let $U\Gamma$ be the underlying set of the ranked set Γ.

Theorem 5.2 μB_Γ is a subtheory of $A_{U\Gamma}$.

Proof: Since μB_Γ is a free theory, we define a morphism of theories $F:\mu B_\Gamma \to A_{U\Gamma}$ by $F(\underline{ap}) = \underline{send}$; if $\Gamma s = n$, $F(s) = \langle s\ x_1 .. x_n\rangle$. F is injective on morphisms. ∎

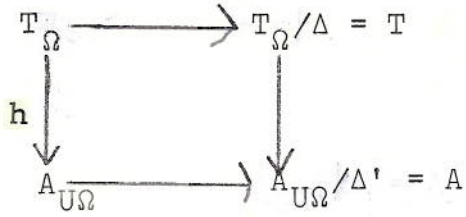
This map fails to be an isomorphism because the actor s may receive any number of messages while in μB it is restricted to precisely Γs arguments.

One reason for interest in actor theories is that they constitute a "universal"(*) class of implementation theories in the following sense:

(*) Here the word "universal" is used in the sense of "universal turing machine" rather than "universal arrow" [30,p.55].

Theorem 5.3 Let T be any theory. Then there is an actor theory A with a theory functor $i:T \to A$.

Proof: Let (Ω,Δ) be a presentation of T, and let $U\Omega$ be the underlying set of Ω. Define a morphism of theories $h:T_\Omega \to A_{U\Omega}$ by $h(s) = \langle s\ e_1 \ldots e_n \rangle$ if $\Omega s = n>0$ and $h(s) = s$ if $\Omega s = 0$. Let $\Delta' = \{(h(f),h(g)) \mid (f,g) \in \Delta\}$, and let $A = A_{U\Omega}/\Delta'$. Now if $f,g \in T_\Omega(k,m)$ and $(f,g) \in E_\Delta$, then $(h(f),h(g)) \in E_{\Delta'}$ (as may easily be verified). Hence the functor $T_\Omega \to A_{U\Omega} \to A_{U\Omega}/\Delta'$ factors through $T_\Omega/\Delta = T$:



■.

Thus, for any theory T, we may create an actor theory A such that every model of A is also a model of T.

6. Conclusions and Relation to other work

6.1 Definitional Interpreters

The approach we have presented is in some measure an outgrowth of the definitional interpreter approach of [24, 28, 38], with sequencing and subtypes removed from the defining language. However, by providing a mathematical semantics for the defining language, we ameliorate the standard objections to metacircular definition while maintaining (we hope) the clarity of the meta-circular style. Furthermore, algebra semantics allows definitions which are not simply transcriptions of meta-circular definitions.

6.2 Initial Algebra Semantics

Another body of work to which algebra semantics is related is that of Goguen, Thatcher, Wagner, and Wright ("the ADJ group"). The idea of their "initial algebra semantics" [13] is to specify a particular Ω-algebra as a "semantic algebra." The unique theory functor from the initial Ω-algebra to the semantic algebra is identified with, say, eval. A priori this seems similar to the approach of Knuth [22] with synthesized attributes only (although inherited attributes can be handled as well [5]).

A major conceptual difference between our scheme and that of [13] (which also uses Reynolds' language as an example) is that we make eval a morphism inside the theory presented by the specification, rather than a functor between

theories. This view avoids any cleverness required to interpret the axioms for eval functorially; indeed, there seems to be no reason to presume that the axioms will be interpretable functorially at all. With our interpretation of eval, a programming language is just an "abstract data type" in the style of [27, 16, 11, 14] in which eval is just another operation.

At this point an issue of philosophy arises. What does a presentation (Ω,Δ) actually present? We hold with Guttag that a presentation specifies a class of models (the (Ω,Δ)-algebras); the theory T_Ω/Δ is a tractable representative of this class. The alternative position, adopted by the ADJ group [14], is that (Ω,Δ) presents a particular algebra, namely the initial (Ω,Δ)-algebra. The major drawback of this view is that it blurs the hard-won distinction between specification and modelling.

Modelling, furthermore, leaves some doubt about which features of the model are required to hold in an implementation. For example, if a model includes some "partially defined" values, which of these are the implementor required to include? Thus a model alone is not sufficient to determine the class of correct implementations. Indeed, distinct equational classes of algebras may have identical initial algebras (e.g. groups and abelian groups both have the one-element group as initial).

A different objection, argued by Lehmann [26] is that

if we are concerned solely with modelling, one can do quite well with much less algebra. The virtue (or saving grace) of the algebraic approach we have espoused, as we see it, is precisely that it is a specification language: it is a logical calculus which delineates the properties which a correct model or implementation must have. We are currently exploring the general relationship between specifications, models, and implementations in a language-independent setting.

An interesting technical difference is the use by the ADJ group of many-typed (or many-sorted) theories, contrasting with our use of single-typed theories. When it is clear what the sorts are, many-sorted theories are attractive. In modelling attribute grammars, for example, one can choose to have one sort per nonterminal [10]. In modelling data types, one can look at theories which are equivalent but not isomorphic (for nondegenerate single-sorted theories, the notions coincide) [11]. On the other hand, in our examples it is not so clear what the sorts should be. Furthermore, the proof theory (E_Δ) for many-typed theories is just the same as ours, except that certain trees are disallowed as wffs. Programs which contain type errors are syntactic errors under a many-sorted regime; in our system the programs run until the error "comes to light" and no deduction is possible. We think of this as a run-time error. A syntactic error which never impedes the course of the program may never

be detected. (This may be either a feature or a bug!). Thus our system is "lazy" in the sense of [8, 18]. A comprehensive treatment would, of course, involve many-sorted theories and a more thorough treatment of errors (e.g. [12, 14]). In this paper we have used only single-sorted theories both for simplicity and to illustrate how errors may be treated in the absence of additional machinery.

A similar contrast occurs in the lambda-calculus between the "static" domain construction of [13] and the "dynamic" construction of domains as retracts [42]. The relation between single-and multiple-typed theories needs to be studied more systematically.

6.3 Denotational Semantics

We may relate the algebraic approach to Scott-Strachey semantics both on philosophical and technical issues.

Philosophically, the Scott-Strachey semantics has been far more concerned with modelling than with specification. Because we are concerned with specification first, we wished to avoid the a priori introduction of specific domains (see [32] for a statement of similar concerns). By choosing the specification language of first-order identities, we were able to construct the domains directly from the equations in a non-creative fashion.

Technically, what we gave up for this was the ability to do unrestricted inductions. Now, algebra semantics is operationally adequate to model every computable function

(via Reynolds' language or by implementing the CUCH directly with axioms like <K **x** y> = x); an important question is whether the semantics gives enough machinery to prove deeper properties of programs without reintroducing some a priori (and possibly operationally irrelevant) ordering structure. Even if the answer is negative, we believe that algebra semantics is worthwhile as an example of a specification language with both well-defined denotational and operational semantics. The method of canonical term algebras [14] is a good start in this direction, indicating how initiality can recapture some of the necessary inductions.

When one adds lattice-theoretic models to the picture, additional questions become askable. One can readily define a fixed-point or paradoxical combinator Y via:

$$\langle x \langle Y\ x\rangle\rangle = \langle Y\ x\rangle$$

but there is no guarantee that the fixed-point so defined is "least", since neither theories nor their algebras have ordering relations imposed on them as yet. For example, one could implement Y as the "optimal fixed point" of Manna and Shamir [31]. If one wishes least fixed points, then additional structure is necessary. This may involve structures such as continuous algebras [13], μclones [45], iteratively closed theories [46], iterative theories [2, 7, 9], or primitive recursive theories [44]. All of these structures are considerably more sophisticated than those considered here, and much mathematics remains to be done before the relations among these various ideas are fully understood.

Acknowledgements

This paper represents a summary of views the author has been expounding since about 1975. We thank K. Indermark for his probing review of an early version of this paper, and Peter Mosses for encouraging discussions. We particularly thank Joe Goguen for his continuing encouragement and his appreciation of our efforts.

References

1. Backus, J.: Programming language semantics and closed applicative languages. Proc. 1st ACM Symp. on Principles of Programming Languages, pp. 71-86. Boston 1973.

2. Bloom, S. L., and Elgot, C. C.: The existence and construction of free iterative theories. JCSS 12:3 (June, 1976).

3. Burstall, R. M., and Darlington, J.: Some transformations for developing recursive programs. Proc. Int'l Conf. on Reliable Software 1975, pp. 465-472.

4. Burstall, R. M., and Goguen, J. A.: Putting theories together to make specifications. Proc. 5th IJCAI 1977, pp. 1045-1058.

5. Chirica, L. M., and Martin, D. F.: An algebraic formulation of Knuthian semantics. Proc. 17th IEEE Symp. on Foundations of Computing 1976.

6. Cohn, P. M.: Universal algebra, New York: Harper-Row 1965

7. Elgot, C. C.: Monadic computation and iterative algebraic theories. Proceedings of the logic colloquium, Bristol 1973,(H. E. Rose & J. C. Shepherdson, eds.), pp. 175-230. North-Holland-Amsterdam 1975.

8. Friedman, D. P., and Wise, D. S.: Cons should not evaluate its arguments. In Automata, Languages and Programming, (S. Michaelson and R. Milner, eds.) Edinburgh University Press: Edinburgh, pp. 257-284, 1976.

9. Ginali, G.: Iterative Algebraic theories, infinite trees, and program schemata. University of Chicago, Ph.D. dissertation, 1976

10. Goguen, J. A.: Semantics of computation. In: Category theory applied to computation and control (E. Manes, ed.), Lecture Notes in Computer Science, Vol. 25, pp. 151-163. Berlin-Heidelberg-New York: Springer 1975.

11. Goguen, J. A.: Correctness and equivalence of data types. In: Mathematical systems theory, Udine, 1975 (G. Marchesihi and S.K. Mitter, eds.), Lecture Notes in Economics and Mathematical Systems, Vol. 131, pp. 352-358. Berlin-Heidelberg-New York: Springer 1976

12. Goguen, J. A.: Abstract errors for abstract data types. Proc. IFIP Working Conference on Formal Description of Programming Language Concepts 1977, St. Andrews, Canada, pp. 21.1-21.32

13. Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B.: Initial algebra semantics and continuous algebras. J. ACM 24, 68-95 (1977)

14. Goguen, J. A., Thatcher, J. W., and Wagner, E. G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types. IBM Yorktown Heights, Research Report RC 6487, 1977

15. Gratzer, G.: Universal algebra. Princeton: Van Nostrand 1968

16. Guttag, J. V., and Horning, J. J.: The algebraic specification of abstract data types. Acta Informatica 10, 27-52 (1978).

17. Guttag, J. V., Horowitz, E., and Musser, D. R.: Abstract data types and software validation. Comm. ACM 21 (1978), 1048-1064.

18. Henderson, P., and Morris, J. H., Jr.: A lazy evaluator. Conf. Rec. 3rd ACM Symp. on Principles of Programming Languages, Atlanta, pp. 95-103. 1976

19. Hewitt, C. E., Bishop, P., and Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. Proc. 3rd IJCAI, pp. 235-245. 1973

20. Hewitt, C. E., and Smith, B.: Towards a programming apprentice. IEEE Trans. on Software Eng SE-1, 26-45 (1975).

21. Hoare, C.A.R.: Proving correctness of data representations. Acta Informatica 1, 271-281 (1972)

22. Knuth, D. E.: Semantics of context-free languages. Math. Sys. Th. 2, 127-145 (1968); correction, 5, 95-96 (1971)

23. Kuhnel, W., Meseguer, J., Pfender, M., and Sols, I.: Primitive recursive algebraic theories with applications to program schemes. (abstract). Cahiers de Topologie et Géométrie Différentielle 16, 271-273 (1975)

24. Landin, P. J.: The mechanical evaluation of expressions. Computer J. 6, 308-320 (1964)

25. Landin, P. J.: The next 700 programming languages. Comm. ACM 9, 157-166 (1966)

26. Lehmann, D. J., and Smyth, M. B.: Data types. University of Warwick, Theory of Computation Report No. 19, May 1977

27. Liskov, B., and Zilles, S.: Specification techniques for data abstractions. IEEE Trans. on Software Eng. SE-1, 7-19 (1975)

28. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Comm. ACM 3, 184-195 (1960)

29. McCarthy, J., et.al.: LISP 1.5 programmer's manual. Cambridge, Mass.: MIT Press 1965

30. Maclane, S.: Categories for the working mathematician. New York: Springer 1971

31. Manna, Z., and Shamir, A.: The theoretical aspect of the optimal fixedpoint. SIAM J. Computing 5, 414-426 (1976).

32. Mosses, P.: Making denotational semantics less concrete. (Extended abstract). Workshop on Semantics of Programming Languages, Bad Honnef, Germany, 21-25 March 1977, University of Dortmund, technical report to appear

33. O'Donnell, M.: Subtree replacement systems: A unifying theory for recursive equations, LISP, Lucid, and combinatory logic. Proc. 9th ACM Symp. on Theory of Computing, Boulder, Colorado, pp. 295-305. 1977

34. Pagan, F. G.: On interpreter-oriented definitions of programming languages. Computer J. 19, 151-155 (1976)

35. Pareigis, B.: Categories and functors. Academic Press 1970

36. Parnas D. L.: A technique for module specification with examples. Comm. ACM 15, 330-336 (1972)

37. Reboh, R. and Sacerdoti, E.D.: A preliminary QLISP manual. Stanford Research Institute, Menlo Park CA, Artificial Intelligence Center Technical Note No. 81, August 1973

38. Reynolds, J. C.: Definitional interpreters for higher-order programming languages. Proc. ACM National Conference 1972, pp. 717-740.

39. Robinson, L., Levitt, K. N., Neumann, P. G., and Saxena, A. R.: On attaining reliable software for a secure operating system. Proc. 1975 Int'l. Conf. on Reliable Software, pp. 267-284.

40. Robinson, L., and Levitt, K. N.: Proof techniques for hierarchically structured programs. Comm. ACM 20, 271-283 (1977)

41. Rosen, B. K.: Tree manipulating systems and Church-Rosser theorems. J. ACM 20, 160-187 (1973)

42. Scott, D.: Data types as lattices. unpublished notes, Amsterdam, 1972

43. Scott, D., and Strachey, C.: Toward a mathematical semantics for computer languages. In: Computers and Automata (J. Fox, ed.), pp. 19-46. New York: Wiley 1972

44. Thatcher, J. W.: Generalized^2 sequential machines. J. Comp. & Sys. Sci. 4, 339-367 (1970)

45. Wand, M.: A concrete approach to abstract recursive definitions. In: Automata, Languages, and Programming (M. Nivat, ed.), pp. 331-345. Amsterdam: North-Holland 1973

46. Wand, M.: Free, iteratively closed categories of complete lattices. Cahiers de Topologie et Géométrie Différentielle 16, 415-424 (1975)

47. Wand, M.: Final algebra semantics and data type extensions. JCSS, to appear.

48. Wand, M.: Algebraic theories and tree rewriting systems. Indiana University, Computer Science Department Technical Report No. 66, July 1977