Final data types and their specification

S. Kamin

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105 78150 Le Chesnay
France
Tél. 954 90 20

Rapports de Recherche

N° 47

360

FINAL DATA TYPES AND THEIR SPECIFICATION

Sam KAMIN

Décembre 1980

FINAL DATA TYPES AND THEIR SPECIFICATION

Sam KAMIN

Abstract

A data type specification is a description of the properties of a data abstraction for the benefit of users and implementers of the abstraction. A data abstraction is a concept having realizations, or implementations, which behave in a certain way.

It is those properties implied by this behavior which we consider essential ; properties specific to some realization are extraneous. The specification problem is : how to present all of the essential properties, and no extraneous ones.

We propose a specification method based upon the notion of "final data type". A final data type is the smallest structure having a given behavior ; every other structure having that behavior maps onto ist homomorphically. This property makes the final data type specification a particularly good source of information about the abstraction it realizes, and eliminates "implementation bias" from the method.

Resumé

Une spécification d'un type de données décrit les propriétés d'une abstraction, et est utilisée par les usagers et les implanteurs de l'abstraction. Une abstraction est un concept dont les réalisations, dites implantations, ont un comportement particulier. Nous considérons comme essentielles les propriétés impliquées par ce comportement et comme non essentielles les autres propriétés (c'est-à-dire, celles qui sont spécifiques à certaines implantations). Le problème en spécification est d'exposer toutes les propriétés essentielles, et aucune propriété non essentielle.

Nous proposons ici une méthode de spécification fondée sur la notion d'algèbre finale. Une algèbre finale est la plus petite structure qui a un certain comportement ; c'est-à-dire, pour chaque structure ayant le même comportement il existe un homomorphisme surjectif à l'algèbre finale. Cette propriété assure que la spécification finale du type rend disponibles les propriétés essentielles, tout en évitant d'influencer l'implantation.

FINAL DATA TYPES AND THEIR SPECIFICATION†

Sam KAMIN

I - INTRODUCTION

Data type encapsulation is by now a widely-accepted method of structuring programs. Abstract use of encapsulated data types requires that a user have not only limited access to elements of the type, but also limited knowledge of its internal details. This in turn requires an implementation-independent presentation of the properties of the data type. Thus it is that data type specification has become an area of active research [4,6,9,10,17,19]. This paper presents a new specification method, and a justification of that method on mathematical grounds.

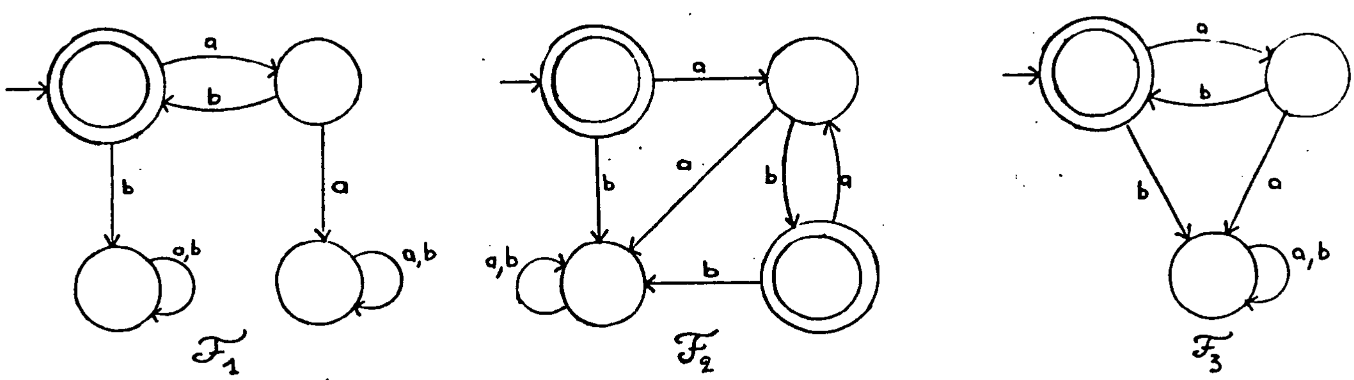
We are, properly speaking, talking about data type extensions, wherein a non-empty collection of known data types is extended by a new type. The new type communicates with the outside world by way of operations returning elements of the known types (as Stack-of-Int communicates via the operations TOP, which returns an element of the known type Int, and ISEMPTY?, which returns a Bool). The new type is a "black box" in that its internal details are hidden ; there are, in general, many ways to fill in the box while achieving the desired function.

---

† The work reported here derives largely from the author's Ph.D. Thesis, presented to the State University of New-York at Stony Brook, and was supported in part by an IBM Graduate Fellowship.

This idea of black boxes is well-known in automata theory. In the way of introduction, we discuss briefly the problem of "specification of finite-state automata". (A similar analysis of this special case is given in [7].)

Suppose we want to specify the class of finite automata accepting the language $(ab)^*=\{\epsilon,ab,abab,\ldots\}$. Each such automaton F has its own internal structure (set of states, next-state function) ; this structure is not important to the user, but it is important to anyone attempting to verify that F indeed accepts $(ab)^*$. Here are three such finite automata :



By a "specification of $(ab)^*$ " we mean a description of some finite automaton which can be used, for example :

- To check whether a given automaton F accepts $(ab)^*$.
- To check whether two strings $u,v\in\{a,b\}^*$ are equivalent, i.e. $\{w/uw\in(ab)^*\}=\{w/vw\in(ab)^*\}$.

We claim that F_3 is inherently a good structure for these purposes, whereas F_1 and F_2 are not :

- Given F, map its states to those of F_3, then prove "strong homomorphism conditions" [11]. (By contrast, try to prove F_2 , using F_1 as the specification.)

- u is equivalent to v iff u and v lead to the same state of F_3. (Using F_1, an equivalence on states must be defined which, in effect, reduces F_1 to F_3.)

Thus, we have in mind an abstraction, $(ab)^*$, which we specify by giving a particular realization among many ; the point is to pick the proper realization. In this case, we find that the proper realization is F_3 ; the reason, of course, is that F_3 is reduced - no two of its states are equivalent. This is not a question of syntax of presentations of automata ; here is an algebraic specification of F_1 :

$$\begin{aligned} \text{ACCEPT?}(\lambda) &= \text{TRUE} \\ \text{ACCEPT?}(b) &= \text{FALSE} \\ \text{ACCEPT?}(a) &= \text{FALSE} \\ \text{ACCEPT?}(aa) &= \text{FALSE} \end{aligned}$$

$$\begin{aligned} ab &= \lambda \\ ba &= b \\ bb &= b \\ aaa &= aa \\ aab &= aa \end{aligned}$$

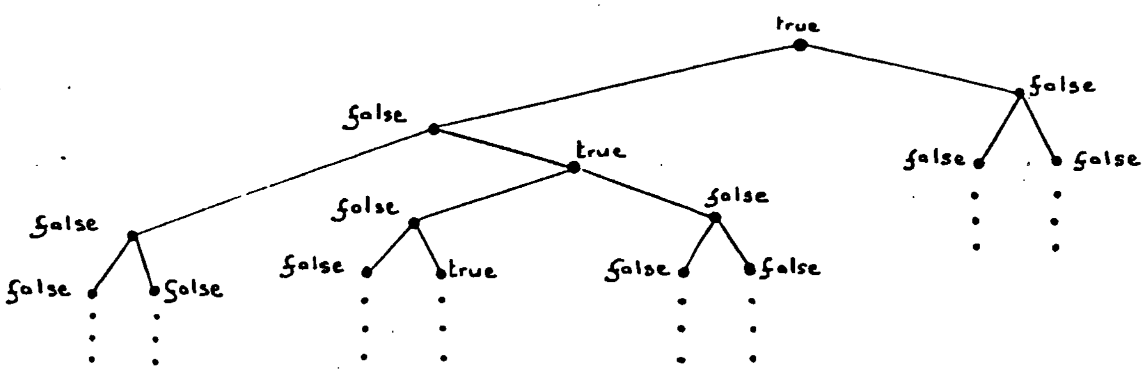
Our comments on the shortcomings of F_1 as a specification apply equally when F_1 is presented in this abstract form.

The problem, then, is how to present reduced finite-state automata. But this can be done in the following way :

Let us suppose that states are infinite binary trees with nodes labelled true or false. Define an accepting state to be a state whose top node is labeled true ; define the next state function of any automaton by

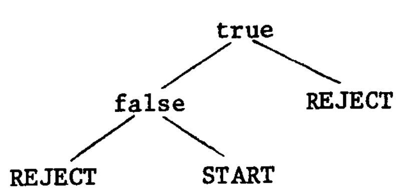
$$\begin{aligned} \text{nsf} : \{a,b\} \times \text{States} &\longrightarrow \text{States} \\ : a, \; {}_{s_1}\overset{t}{\wedge}{}_{s_2} &\longmapsto s_1 \\ b, \; {}_{s_1}\overset{t}{\wedge}{}_{s_2} &\longmapsto s_2. \end{aligned}$$

If such an automaton is to accept $(ab)^*$, it must have as its start state :

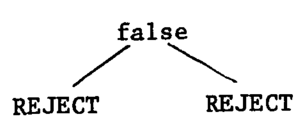


This in turn determines what its other states will be. The only problem now is to describe the start state finitely ; but this is easily done using a recursive definition :

START =

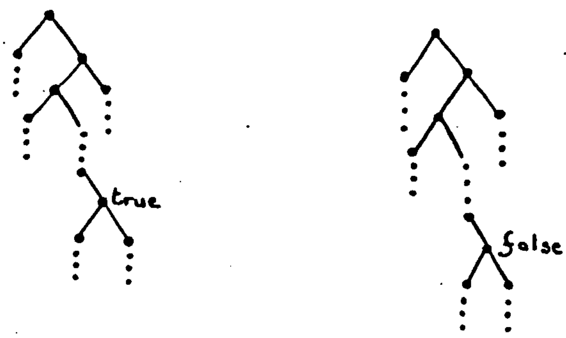


where REJECT =



<u>Theorem</u> : Any automaton with infinite trees as states, and with the next-state function nsf, is reduced.

<u>Proof</u> : Suppose two distinct states are equivalent. Since they are different states, they must have different labels at some node.



Then the appropriate string bab...b is accepted by the first state and not by the second. Thus, these states are not equivalent, which is a contradiction. □

Therefore, our automaton with start state START -since it does accept $(ab)^*$- is isomorphic to F_3. Its three states are START, REJECT, and

```
       false
      /     \
REJECT       START .
```

Using the specification requires knowledge of infinite trees, recursive functions, etc. But we gain the advantages of having specified a reduced automaton :

- To prove that F_1 accepts $(ab)^*$, map its states onto infinite trees and prove the homomorphism conditions.
- To prove, for example, b≡aa :

START after input aa

```
=        false          after input a
        /     \
   REJECT      START
```

= REJECT

= START after input b.

Thus, we can exploit the mathematical relationship between the reduced finite-state automaton and other automata accepting the language. The question is : can the idea of reduced automata be generalized to data types, and can the specification method suggested here be applied ?†

The next section of this paper contains the definitions of various algebraic concepts, including many-sorted algebra, data type extension, and final data type, which is the generalization of reduced automaton. We refer the reader to [6] for a fuller treatment of many-sorted algebras. Section III extends the idea of specifying reduced automata to specifying final data types, and section IV shows how specifications can be used. Finally, section V draws some conclusions from our work.

II - DEFINITIONS

This section is devoted to technical definitions. Although some motivations and examples are given, fuller discussions will be found in the references [6, 9, 14, 17]. See in particular [6], from which our notation is drawn.

The programmer uses a data type via certain operations, which are symbols that name functions ; that is, there is a syntax for using the type, and a semantics established either by the language or by an encapsulated type definition. Signature and algebra are the formal versions of syntax and semantics, respectively.

Definition : An S-sorted signature Σ is a family of sets $<\Sigma_{w,s}>_{w\in S^*, s\in S}$. S is called the set of sorts, Σ the set of operators. To denote that $\sigma\in\Sigma_{w,s}$ we will sometimes write $\sigma: w\to s$ (Σ being understood from context) or $\sigma:\to s$ when $w=\epsilon$, the empty sequence in S^*. □

† The reader may wonder why the expression "$(ab)^*$" is not taken as the specification, as well it could be. The problem is that this method depends upon the particular relationship between finite automata and regular expressions, and therefore cannot be generalized.

Definition : Given an S-sorted signature Σ, an S-sorted Σ-algebra A is a collection of sets $\langle A_s \rangle_{s \in S}$ and an assignment to each $\sigma : s_1 \ldots s_m \to s$ of a function

$$\sigma_A : A_{s_1} \times \ldots \times A_{s_m} \to A_s . \qquad \square$$

Being interested in the situation where one type is being added to a pre-defined collection of types, we want slightly modified versions of these definitions.

Definition : An S-sorted data type signature is a pair $\langle \Sigma, s_N \rangle$, with Σ an S-sorted signature, $s_N \in S$. s_N is the name of the "new type" or "type of interest"; elements of $S \backslash \{s_N\}$ are called "known" or "pre-defined" types. $\square$

Definition : A $\langle \Sigma, s_N \rangle$ - data type extension (or just data type), for $s_N \in S$, is a Σ-algebra. $\square$

When Σ-algebra D is referred to as a $\langle \Sigma, s_N \rangle$-data type, it is just to emphasize the newness of s_N. This definition could easily be extended to allow a set of simultaneously-defined new types.

Example

Consider a new type multiset, i.e. set with repetitions. There are predefined types Atoms and Nat, an operation to find the number of occurrences of any atom in a multiset, plus several functions for building multisets. Thus,

$$S = \{\text{MSet, Atoms, Nat}\}$$

Σ_{MSet} consists of

NULL : $\to$ MSet
SINGLE : Atoms $\to$ MSet
UNION : MSet $\times$ MSet $\to$ MSet
REMOVE : Atoms $\times$ MSet $\to$ MSet
COUNT : Atoms $\times$ MSet $\to$ Nat .

Naturally, we are interested in $<\Sigma_{MSet}$,MSet>-data types ; this data type signature is pictured in Figure 1. We will also assume that in any $<\Sigma_{MSet}$,MSet>-data type D,

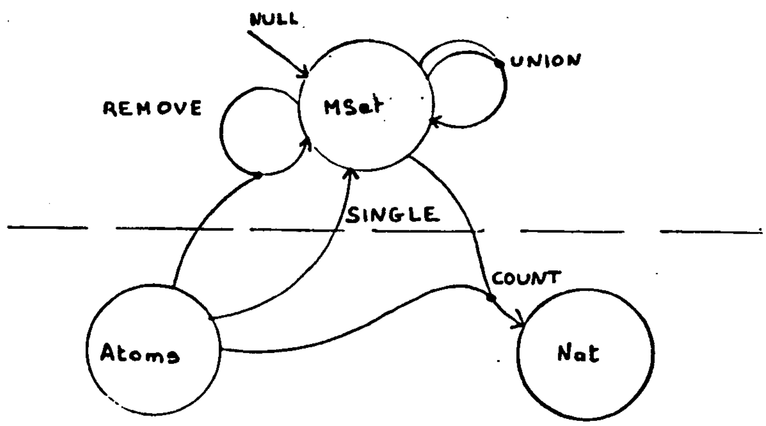


FIGURE 1

$$D_{Atoms} = \{a_0, a_1, a_2, \ldots\}$$

and

$$D_{Nat} = \{0, 1, 2, \ldots\}.$$

Still, there are many possibilities for D_{MSet} and for the functions $NULL_D$, $SINGLE_D$, etc. Most choices would not be at all consistent with our intuition of how MSet -as a "black box"- should behave. Here is one set of definitions which is :

Let the $<\Sigma_{MSet}$,MSet>-data type M have M_{Atoms} and M_{Nat} as above, and let $M_{MSet} = (M_{Atoms})^*$. Then define :

NULL : $\mapsto \epsilon$

SINGLE : i $\mapsto$ <i> , the one element sequence containing i.

UNION : v,w $\mapsto$ vw

REMOVE : i,w $\mapsto$ subsequence of w obtained by removing all occurrences of i.

COUNT : i,w $\mapsto$ # of occurrences of i in w. □

From the "black box" point of view, the elements of M_{MSet} can be observed by performing COUNT operations, but cannot be manipulated directly. (For example, the user cannot take the head or tail of an element of M_{MSet}, even though it is a sequence.) Although this view is quite reasonable, and has been taken before [1,9,17,26], it is not correct in certain circumstances : for example, when one is starting "from scratch". In those cases, the analysis of this paper is not applicable.

Definition : If $\langle\Sigma, s_N\rangle$ is a data type signature, the operators of Σ divide themselves into three classes :

- $\bigsqcup\{\Sigma_{w,s_N} / w \in S^*\}$ is the set of input operators.
- $\bigsqcup\{\Sigma_{w,s} / s \neq s_N$, w contains $s_N\}$ is the set of output operators.
- The remaining "old" operators which we generally take for granted. □

Example

The input operations of $\langle\Sigma_{MSet}, MSet\rangle$ are {NULL, SINGLE, UNION, REMOVE} and the output operations are {COUNT}. No old operations are mentioned. □

Definition : A homomorphism $h: A \rightarrow B$ for S-sorted Σ-algebras A and B is a collection of maps $h = \langle h_s\rangle_{s \in S}$, where for each s, $h_s : A_s \rightarrow B_s$, such that for each $\sigma : s_1 \ldots s_m \rightarrow s$,

$$\forall a_1 \in A_{s_1}, \ldots, a_m \in A_{s_m},$$

$$h_s(\sigma_A(a_1, \ldots a_m)) = \sigma_B(h_{s_1}(a_1), \ldots, h_{s_m}(a_m)).$$

If every h_s is a bijection, then h is an isomorphism. □

A well-known example of a particular Σ-algebra is the word algebra, or free algebra, of Σ. This has as its elements the (properly typed) expressions which are formed from Σ. From this definition, we will then define the important concept of a "derived operator". Again, more explanation will be found in the references.

Definition : The Σ-algebra T_Σ has as its elements the strings formed from symbols in Σ, plus "(", ")", and "," , in the following way :

- $"\sigma" \in T_{\Sigma,s}$ whenever $\sigma \in \Sigma_{\epsilon,s}$.
- $"\sigma(t_1,\ldots,t_m)" \in T_{\Sigma,s}$ whenever $\sigma \in \Sigma_{s_1\ldots s_m,s}$ and $"t_1" \in T_{\Sigma,s_1},\ldots,"t_m" \in T_{\Sigma,s_m}$.

Thus, the elements of T_Σ are just strings ; the operators take strings to strings, as follows :

- If $\sigma \in \Sigma_{\epsilon,s}$, then $\sigma_{T_\Sigma} : \mapsto "\sigma"$
- If $\sigma \in \Sigma_{s_1\ldots s_m,s}$, then $\sigma_{T_\Sigma} : "t_1",\ldots,"t_m" \mapsto "\sigma(t_1,\ldots,t_m)"$. □

The intuitive idea of evaluating an expression is made precise by :

Theorem : For any Σ-algebra A, there is a unique homomorphism $eval_A : T_\Sigma \rightarrow A$. □

Now, suppose we have a new symbol "$\times$", which is to act as a variable of sort s. An expression of sort s' with one or more occurrences of $\times$ can be regarded as a function from A_s to $A_{s'}$, for any Σ-algebra A. This is justified by :

Theorem : Let $\Sigma(\times)$ denote the signature which is the same as Σ except that $\Sigma(\times)_{\epsilon,s} = \Sigma_{\epsilon,s} \cup \{\times\}$. Furthermore, call $T_\Sigma(\times)$ the Σ-algebra with carriers those of $T_{\Sigma(\times)}$, but with operators Σ. (That is, use $\times$ to form expressions, but then forget that it is an operator.) Then, given Σ-algebra A and $a \in A_s$, there is a unique homomorphism

$$eval_A[a] : T_\Sigma(\times) \rightarrow A$$

such that $eval_A[a]_s(\times) = a$. □

In other words, once we know the value of x, we know the value of any expression containing x. Thus, a term of sort s' with variable x uniquely determines a function from A_s to $A_{s'}$. Such terms are called "derived operators" and are written $t[x]$; then for $a \in A_s$, we write $t[a]$ instead of $eval_A[a]_{s'}(t)$.

We are only interested in those elements of an algebra which are denotable by expressions. Since this also greatly simplifies the mathematics, we assume from now on that all algebras -unless stated otherwise- are prime :

Definition : A Σ-algebra A is prime if $eval_{A,s}$ is onto for each $s \in S$. □

Lemma : Given two prime Σ-algebras A and B, if there is a homomorphism $h:A \rightarrow B$, then it is unique and onto.

Proof : We must have $h \circ eval_A = eval_B : T_\Sigma \rightarrow B$, since $eval_B$ is unique. Since $eval_A$ is onto A, h is uniquely determined ; since $eval_B$ is onto B, h is also. □

On the other hand, implementations do not, in general, give rise to prime algebras ; nor do our specifications. In both cases, it is sometimes necessary to isolate the range of eval by a "representation invariant " [12].

We are finally in a position to say what we mean by two data types having the same behavior as "black boxes".

Definition : Two $<\Sigma, s_N>$-data types D and E are interchangeable if

- "The pre-defined types are equal."
  That is, letting $S' = S \backslash \{s_N\}$, Σ' = old operations of Σ, D' (resp. E') = the S'-sorted Σ'-algebra obtained by forgetting D_{s_N} (resp. E_{s_N}) and all input and output operations, we have $D' \cong E'$.

- "Output operations work correctly."
  That is, $eval_{E,s} = i \circ eval_{D,s}$ for all $s \neq s_N$, where i is the isomorphism from D' to E'. (We have assumed that D' and E' are prime ; although this does not follow from the primeness of D and E, it is always true in practice, because of the hierarchical nature of data type extension.)

Example

Consider the $<\Sigma_{MSet}$,MSet>-data type M defined earlier. We have

$$eval_{M,Nat} : T_{\Sigma_{Mset},Nat} \longrightarrow M_{Nat}$$

$$: \text{"COUNT}(a_0,\text{NULL})\text{"} \longmapsto 0$$

$$\text{"COUNT}(a_0,\text{SINGLE}(a_0))\text{"} \longmapsto 1$$

$$\text{"COUNT}(a_1,\text{UNION}(\text{SINGLE}(a_1),\text{SINGLE}(a_1)))\text{"} \longmapsto 2$$

$$\vdots$$

Any other $<\Sigma_{MSet}$,MSet>-data type M' will be interchangeable with M as long as M'_{Atoms} and M'_{Nat} are (isomorphic to) M_{Atoms} and M_{Nat} respectively, and $eval_{M',Nat}$ is the same function as $eval_{M,Nat}$. ($eval_{M',Atoms}$ can be ignored here, since $T_{\Sigma_{Mset},Atoms}$ contains no new expressions.)

For example, suppose M' is identical to M except that

$$\text{UNION}_{M'} : v,w \longmapsto wv.$$

This minor change leads to a non-isomorphic, but interchangeable, algebra.

Or, let M'' have M''_{MSet} = {functions $f:M''_{Atoms} \longrightarrow M''_{Nat} / f(a_i)=0$ for all but a finite number of a_i}

and define

$$\mathrm{NULL}_{M''} : \;\mapsto \lambda a_i.0$$

$$\mathrm{SINGLE}_{M''} : a_i \mapsto \lambda a_j.\ \underline{\text{if}}\ a_i = a_j\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0$$

$$\mathrm{UNION}_{M''} : f_1, f_2 \mapsto \lambda a_i . f_1(a_i) + f_2(a_i)$$

$$\mathrm{REMOVE}_{M''} : a_i, f \mapsto \lambda a_j.\ \underline{\text{if}}\ a_i = a_j\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ f(a_j)$$

$$\mathrm{COUNT}_{M''} : a_i, f \mapsto f(a_i).$$

The reader may verify that this data type extension is interchangeable with M, and also that there is a homomorphism from M to M''. □

We want the data abstraction to be simply "that thing which we implement". So we take it to be the class of all data types having the correct behavior :

Definition : The $<\Sigma, s_N>$-data abstraction $A(D)$ realized by $<\Sigma, s_N>$-data type D is defined to be the class of all data types E which are interchangeable with D. A is a data abstraction if it is equal to $A(D)$ for some D, and D is said to "realize" A. □

Definition : Given a data abstraction A, data type F is final in A if, for any other $D \in A$, there exists a homomorphism $\underline{abs}_D : D \rightarrow F$. □

We will see that this homomorphism can be regarded as associating with an element of D the "abstract" element it represents. Moreover, the function will be used as an "abstraction function" [12] for proving implementations.

Theorem : [5,27] Every data abstraction contains a unique (up to isomorphism) final data type F.

Proof : We can construct F in the following way : Let $D \in A$ be arbitrary, and consider the family of equivalences $\equiv \; = \; <\equiv_s>_{s \in S}$ on D, with

$\equiv_s$ trivial (i.e. just equality) for $s \neq s_N$, and $\equiv_{s_N}$ given by :

$$d \equiv_{s_N} d' \quad \text{if for any derived operator } t[\times]: D_{s_N} \to D_s \text{ , for } s \neq s_N, \; t[d]=t[d'].$$

(We say that d and d' are <u>indistinguishable</u>.)

It is obvious that $\equiv$ is a family of equivalence relations ; we need that it be a congruence on D – that is, that $d_1 \equiv_{s_1} d_1', \ldots, d_m \equiv_{s_m} d_m'$ $\Longrightarrow \sigma_D(d_1,\ldots,d_m) \equiv_s \sigma_D(d_1',\ldots,d_m')$ for all $\sigma : s_1\ldots s_m \to s$. We consider two cases :

- $s \neq s_N$. Then $\sigma_D(d_1,\ldots,d_m) = \sigma_D(d_1', d_2,\ldots,d_m) = \ldots = \sigma_D(d_1',\ldots, d_m')$, so $\sigma_D(d_1,\ldots, d_m) \equiv_s \sigma_D(d_1',\ldots, d_m')$. Each step is justified because either $s_i \neq s_N$, so $d_i = d_i'$ and the expression is unchanged, or $s_i = s_N$, in which case $\sigma(d_1',\ldots, d_{i-1}',\times, d_{i+1},\ldots,d_m) : D_{s_N} \to D_s$ is a derived operator[†], and the step follows from the definition of $\equiv_{s_N}$.

- $s=s_N$. Then for any derived operator $t[\times] : D_{s_N} \to D_{s'}$, $s' \neq s_N$,

$$\begin{aligned} &t[\sigma_D(d_1, d_2,\ldots, d_m)] \\ &\quad = t[\sigma_D(d_1', d_2,\ldots, d_m)] \\ &\quad = \ldots \\ &\quad = t[\sigma_D(d_1',\ldots, d_m')] \\ &\Rightarrow \sigma_D(d \;,\ldots, d_m) \equiv_{s_N} \sigma_D(d_1',\ldots, d_m') \; , \end{aligned}$$

each step being justified very much as above.

The claim is that the algebra $D/\equiv$, having carriers $(D/\equiv)_s = D_s/\equiv_s$,

---

† More properly, it can be represented as a derived operator by finding expressions $\overline{d_1'},\ldots,\overline{d_{i-1}},\overline{d_{i+1}'},\ldots,\overline{d}_m$ with values $d_1',\ldots,d_m$. Such expressions exist by primeness of D.

and functions $\sigma_{D/\equiv}([d_1],\ldots,[d_m]) = [\sigma_D(d_1,\ldots,d_m)]$, is final. First, $D/\equiv$ is interchangeable with D :

- $D_s \cong (D/\equiv)_s$ for all $s \neq s_N$, since $\equiv_s$ is the identity.
- $eval_{D,s} = i_\circ eval_{D/\equiv,s}$ for all $s \neq s_N$, where $i_s : D_s \longrightarrow (D/\equiv_s) : d \longmapsto [d]_\equiv$. This is, of course, an isomorphism.

Most importantly, we want to show that $D/\equiv$ is final, so suppose $E \in A$ and consider the function

$$\underline{abs}_E : E_{s_N} \longrightarrow (D/\equiv)_{s_N}$$

$$: e \longmapsto [eval_{D,s_N}(t)]_\equiv \text{ , where } t \in T_{\Sigma,s_N} \text{ is some expression such that } eval_{E,s_N}(t) = e.$$

Such a t must exist, since E is prime ; the problem is to show that abs is well-defined. But if t and t' are distinct terms with $eval_{E,s_N}(t) = eval_{E,s_N}(t') = e$, then $eval_{D,s_N}(t) \equiv_{s_N} eval_{D,s_N}(t')$, since both have the same behavior as e. Thus, $[eval_{D,s_N}(t)]_\equiv = [eval_{D,s_N}(t')]_\equiv$, and abs is well-defined.

It remains to be shown only that $\underline{abs}_E$ is a homomorphism ; that is, that for all $e_1 \in E_{s_1},\ldots,e_m \in E_{s_m}$,

$\underline{abs}_E(\sigma_E(e_1,\ldots,e_m)) = \sigma_{D/\equiv}(\underline{abs}_E(e_1),\ldots,\underline{abs}_E(e_m))$.
But if $t_1,\ldots,t_m \in T_\Sigma$ are such that $eval_E(t_i) = e_i$, $1 \le i \le m$, then $eval_E(\text{"}\sigma(t_1,\ldots,t_m)\text{"}) = \sigma_E(e_1,\ldots,e_m)$, so

$$\begin{aligned}\underline{abs}_E(\sigma_E(e_1,\ldots,e_m)) &= eval_{D/\equiv}(\text{"}\sigma(t_1,\ldots,t_m)\text{"}) \\ &= \sigma_{D/\equiv}(eval_{D/\equiv}(\text{"}t_1\text{"}),\ldots,eval_{D/\equiv}(\text{"}t_m\text{"})) \\ &= \sigma_{D/\equiv}(\underline{abs}_E(e_1),\ldots,\underline{abs}_E(e_m)).\end{aligned}$$

Finally, if F and F' are both final, then there exist onto homomorphisms in both directions, so they are isomorphic. □

The theorem depends upon the assumed primeness of all algebras in A (and is, indeed, false otherwise). Results on final algebras in larger categories (containing non-prime algebras) appear in [2,3].

Corollary to proof : D is final in A(D) iff no two elements of D_{s_N} are indistinguishable. □

The final data type is, in a sense, the "smallest" data type having the desired behavior. It is obtained by identifying as many elements as could reasonably be identified. As such, it often coincides with the intuitive picture of the data abstraction.

Examples

(1) M is not final. Consider, for example, a_0a_1 and $a_1a_0 \in M_{MSet}$. There is no way to distinguish these elements from the outside.

On the other hand, any two functions f_1 , $f_2 \in M''_{MSet}$ are distinguishable. By definition, if f_1 and f_2 are distinct, there exists $a_i \in M''_{Atoms}$ with $f_1(a_i) \neq f_2(a_i)$. Thus, f_1 and f_2 are distinguished by the derived operator COUNT(a_i,x).

(2) Finite-state automata can be regarded as S_{FA}-sorted Σ_{FA}-algebras, with $S_{FA} = \{$Bool, States$\}$ and Σ_{FA} given by

$$\begin{aligned} \text{START} &: \longrightarrow \text{States} \\ a,b &: \text{States} \longrightarrow \text{States} \\ \text{ACCEPT?} &: \text{States} \longrightarrow \text{Bool}. \end{aligned}$$

The $<\Sigma_{FA}$, States$>$- data types which interest us are those data types F such that $F_{Bool} = \{$true, false$\}$ and ACCEPT? $(c_n(c_{n-1}(\ldots(c_1(\text{START}))\ldots)$ = true if and only if $c_1c_2\ldots c_n \in (ab)^*$. This still leaves considerable freedom to choose F_{States}, ranging from

$$F_{States} = \{a,b\}^*$$

with ACCEPT? : $w \mapsto$ true if $w \in (ab)^*$, false, o.w.

to

$$F'_{States} = \{r_1, r_2, r_3\}$$

with ACCEPT? : $r_1 \mapsto$ true; $r_2, r_3 \mapsto$ false

$$\begin{aligned} \text{START} &: \ \mapsto r_1 \\ a &: r_1 \mapsto r_2 \\ &\quad r_2, r_3 \mapsto r_3 \\ b &: r_2 \mapsto r_1 \\ &\quad r_1, r_3 \mapsto r_3. \end{aligned}$$

F', corresponding to the reduced automaton, is the final data type here. □

III - SPECIFYING FINAL DATA TYPES

The key to our approach is to concentrate on operations which observe elements of the data type, rather than on those which build elements.

Definition : A distinguishing term for a data type signature $<\Sigma,s_N>$ is an element of $T_\Sigma(\times)_s$, with $\times$ a variable of sort s_N and $s \neq s_N$ (i.e. a derived operator from s_N to s). □

Distinguishing terms may be used to observe the difference between two elements of the new type. Whether this succeeds or not will depend upon the term and the data type ; for example, COUNT(a_2 , $\times$) will distinguish between some pairs of multisets, and not between others.

Definition : A distinguishing set DS for a $<\Sigma,s_N>$-data type D is a set of distinguishing terms such that for any two distinguishable elements d_1 , $d_2 \in D_{s_N}$, there is a $t[\times] \in DS$ such that $t[d_1] \neq t[d_2]$. □

That a distinguishing set exists for every data type is clear ; the set of all distinguishing terms is one. Notice also that a d.s. for D is also a d.s. for any other data type in $A(D)$.

Examples

(1) $\{COUNT(a,\times)/a \in M_{Atoms}\}$ is a d.s. for M.
(2) {ACCEPT?($\times$)} is not a distinguishing set for our finite-state automata. One d.s. is {ACCEPT?($\times$), ACCEPT?(b($\times$))} ; of course, any superset of this set is also a d.s. □

The problem is the following : Given an algebra D of known types, we want to add the new type s_N. This involves deciding upon a representation of D_{s_N} , and then defining the new operations over that representation. The difficulty is that we require the algebra so obtained to be final.

The trick is to recognize that the statement that DS is a distinguishing set amounts to saying that the elements of the new type are characterized by their values at each of the derived operators in DS. Then we can simply take the representation of an element to be the collection of all its values for all these functions. Thus, proceed as follows :

1. Suppose the DS is $\{t_i[\times]/i\in I\}$.
2. Take the representation of the new type to be $D_{s_N} = \prod_{i\in I} D_{s_i}$, where s_i is the sort of $t_i[\times]$.
3. Define all the new operations of the data type as functions over D_{s_N}.
4. For all $i\in I$, prove that the meaning of $t_i[\times]$, as given by step 3, is Π_i, the i-th projection of D_{s_N}.

Example

From the DS {ACCEPT?(×), ACCEPT? (b(×))}, we obtain representation States = Bool × Bool. We now define the operators.

$$
\begin{aligned}
\text{ACCEPT?} &: \text{States} \longrightarrow \text{Bool} \\
&: \quad s \longmapsto \Pi_1(s) \\
a &: \text{States} \longrightarrow \text{States} \\
&: \quad s \longmapsto \langle \text{false}, \Pi_1(s)\rangle \\
b &: \text{States} \longrightarrow \text{States} \\
&: \quad s \longmapsto \langle \Pi_2(s), \Pi_1(s)\wedge\Pi_2(s)\rangle \\
\text{START} &: \quad \longrightarrow \text{States} \\
&: \quad \longmapsto \langle \text{true}, \text{false}\rangle.
\end{aligned}
$$

We easily do step 4 :

$$
\begin{aligned}
&- \text{ACCEPT?}(s) = \Pi_1(s). \\
&- \text{ACCEPT?}(b(s)) = \Pi_1(\langle \Pi_2(s), \Pi_1(s)\wedge\Pi_2(s)\rangle) \\
&\qquad = \Pi_2(s).
\end{aligned}
$$

□

Now, any two different states must differ in either their first or second projections, so by applying either ACCEPT?(×) or ACCEPT?(b(×)), they can be distinguished. Therefore the data type we have defined , or rather its prime part , is final. It would have been possible to give an <u>incorrect</u> definition - i.e., one not accepting $(ab)^*$ - but not a non-final one.

The problem with this "method" is that it involves infinite sets at each step. It turns out that we can give finite descriptions of d.s.' s, which in turn lead to finite descriptions of the set D_{s_N} and a finite number of operations to define and prove. The key idea is to use function spaces and recursive domain definitions to avoid the infinite cross product in step 2.

The four steps now become :

1. Give a d.s. in the following form :

$$DS(s_N) = \{t_1[x]/x_1^1 \in s_1^1, \ldots, x_{n_1}^1 \in s_{n_1}^1\}$$
$$\cup \ \vdots$$
$$\cup \{t_k[x]/x_1^k \in s_1^k, \ldots, x_{n_k}^k \in s_{n_k}^k\},$$

where x is a variable of sort s_N, $x_1^i, \ldots, x_{n_i}^i$ are all the variables of t_i other than x, and furthermore :

- For at least one j, the sort of $t_j[x]$ is not s_N, and
- None of $s_1^1, \ldots, s_{n_1}^1, \ldots, s_1^k, \ldots, s_{n_k}^k$ is s_N.

The first requirement is justified by the intuition that $DS(s_N)$ really describes the set of distinguishing terms formed by composition of $t_1, \ldots, t_k$. No distinguishing terms could be obtained if all of $t_1, \ldots, t_k$ had sort s_N. The second requirement is technical ; although a DS such as

$$DS(s_N) = \{=(x,y)/y \in s_N\}$$

seems intuitively reasonable, it is not clear to this author how to use it.

2. The representation of D_{s_N} is the smallest non-trivial solution to the domain equation :

$$(*) \quad D_{s_N} = (D_{s_1^1} \times \ldots \times D_{s_{n_1}^1} \longrightarrow D_{s_1})$$
$$\times \ \vdots$$
$$\times (D_{s_1^k} \times \ldots \times D_{s_{n_k}^k} \longrightarrow D_{s_k}),$$

where s_i is the sort of $t_i[x]$, and "$\longrightarrow$" is the function space constructor (i.e. $A \longrightarrow B$ is the set of all functions from A to B). Since s_i may be s_N for some i, this equation can be non-trivial ; we discuss solutions to (*) in the appendix.

3. Define the new operations over D_{s_N}.

4. For each i, prove that, in the Σ-algebra D defined in steps 2 and 3 , we have

$$t_i \left| \begin{matrix} d_1^i, \ldots, d_{n_i}^i \\ x_1^i, \ldots, x_{n_i}^i \end{matrix} \right. [d] = \Pi_i \, (d) \, (d_1^i, \ldots, d_{n_i}^i),$$

for all $d \in D_{s_N}$, $d_j^i \in D_{s_j^i}$. $t_i \left| \begin{matrix} d_1^i, \ldots, d_{n_i}^i \\ x_1^i, \ldots, x_{n_i}^i \end{matrix} \right.$ is the derived operator resulting from substituting d_j^i for variable x_j^i in t_i. † If, as often occurs in practice, the term $t_i[x]$ has the form $\sigma(x,x_1,\ldots,x_m)$, a single function symbol with one argument of sort s_N, we can take this requirement to be the <u>definition</u> of operation σ, thus avoiding one definition and one proof.

<u>Examples</u>

1. The last example is already almost in the required form :
   DS(States) = {ACCEPT?(x)}∪{ACCEPT?(b(x))}.

2. An alternative DS for finite-state automata is :
   DS(States) = {ACCEPT?(x)}∪{a(x)}∪{b(x)}.

   This leads to the domain equation

   States = Bool × States × States,

   which is exactly the tree representation given in the introduction. All three operators in the DS fall into the special category of a single function symbol with variable, so we take the definitions of ACCEPT? , a , and b immediately to be Π_1 , Π_2 ; and Π_3 . It remains to define the operation START, which we do exactly as in the Introduction. The complete specification is given in Figure 2.

3. DS(MSet) = {COUNT(a,x)/a∈Atoms}, from which we get

   MSet = (Atoms ⟶ Nat)

   and COUNT : a,s ⟼ s(a).

   It remains to define the other operations, which we do exactly as for M''. The complete specification is given in Figure 3. □

---

† More properly, the result of substituting expression $\bar{d}_j^i \in T_{\Sigma,s_j^i}$, whose value is d_j^i, for variable x_j^i.

A number of other examples appear in [15].

Data type "Finite automata accepting $(ab)^*$"

S_{FA} = {States, Bool}

Σ_{FA} = START : $\longrightarrow$ States

a,b : States $\longrightarrow$ States

ACCEPT? : States $\longrightarrow$ Bool

DS (States) = {ACCEPT?(x)}∪{a(x)}∪{b(x)}.

START : $\longmapsto$ <true, <false, REJECT, START>, REJECT>,

where REJECT : $\longmapsto$ <false, REJECT, REJECT>.

Figure 2

Data type "Multisets"

S_{MSet} = {MSet , Atoms , Nat}

Σ_{MSet} = NULL : $\longrightarrow$ MSet

SINGLE : Atoms $\longrightarrow$ MSet

UNION : MSet × MSet $\longrightarrow$ MSet

REMOVE : Atoms × MSet $\longrightarrow$ MSet

COUNT : Atoms × MSet $\longrightarrow$ Nat

DS(MSet) = {COUNT(a,x)/ a∈ Atoms}

NULL : $\longmapsto \lambda a_i$: Atoms.0

SINGLE : $a_i \longmapsto \lambda a_j$: Atoms. <u>if</u> $a_i = a_j$ <u>then</u> 1 <u>else</u> 0

UNION : $f, f' \longmapsto \lambda a_i$: Atoms. $f(a_i) + f'(a_i)$

REMOVE : $a_i, f \longmapsto \lambda a_j$: Atoms. <u>if</u> $a_i = a_j$ <u>then</u> 0 <u>else</u> $f(a_j)$.

Figure 3

IV - APPLICATIONS

Our basic argument is that the final data type is the best concrete representative of a data abstraction. In several illustrations of the use of final data type specifications, we will show how the promise of this argument is fulfilled. It is particularly interesting to contrast these specifications with algebraic specifications [6, 9, 10, 17, 19], which are not constrained to define only final data types. One connection should be mentioned now : Given a consistent, sufficiently-complete [9] axiomatization A of a data abstraction $\mathcal{A}$, the final algebra of $\mathcal{A}$ is just the initial algebra of the maximal consistent extension of A.

Properties of Data Types

1. Axioms

Suppose we are given an algebraic specification of multisets :

COUNT (×, NULL) = 0
COUNT (×, SINGLE(y)) = if ×=y then 1 else 0
COUNT (×, UNION(s,s')) = COUNT(×,s) + COUNT(×,s')

REMOVE (×,NULL) = NULL
REMOVE (×,SINGLE(y)) = if ×=y then NULL else SINGLE(y)
REMOVE (×,UNION(s,s')) = UNION(REMOVE(×,s), REMOVE(×,s')).

Intuitively, this specification is sufficient and correct, and it does satisfy the technical conditions of consistency and sufficient completeness [9].

On the other hand, we can see that the two terms UNION(s,s') and UNION(s',s), for any distinct s,s' , are not equated. If they are indistinguishable, then they must be equal in the final data type. So, referring back to the final specification (Figure 3), and using knowledge of predefined types, λ-abstraction, and so on, we attempt to verify this :

$$\begin{aligned} \text{UNION}(s,s') &= \lambda a_i .\ s(a_i) + s'(a_i) \\ &= \lambda a_i .\ s'(a_i) + s(a_i) \\ &= \text{UNION}(s',s). \end{aligned}$$

2. Finding Constructors

It is often convenient, when working with a data type, to know that a certain subset of all the expressions generates all elements of the type. (That is, that $T \subseteq T_{\Sigma,s_N}$ is such that $\text{eval}_{D,s_N}|_T$ is onto D_{s_N}.) For example, it is easy to show, from either the algebraic or final data type specification, that every multiset is the value of an expression consisting only of NULL, SINGLE and UNION operations. On the other hand, it is hard to use the algebraic specification to show that each multiset is the value of a term of the form

$$\text{UNION}(\text{SINGLE}(a_{i_1}), \text{UNION}(\text{SINGLE}(a_{i_2}), \ldots, \text{UNION}(\text{SINGLE}(a_{i_k}), \text{NULL}))\ldots),$$

where $i_1 \le i_2 \le \ldots \le i_k$. (The problem is that relationships among the constructors SINGLE, UNION, and NULL are not given ; even

SINGLE(a) = UNION(SINGLE(a),NULL) is not directly demonstrable from the axioms.) In the final data type, this is true, and we can use the final data type specification to show it. Thus, by induction on terms, considering only NULL, SINGLE and UNION, we have :

- NULL - nothing to prove,
- SINGLE(a_i). But SINGLE(a_i) =

$$\lambda a_j . \underline{\text{if}}\ a_i = a_j\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0$$
$$= \lambda a_j . (\underline{\text{if}}\ a_i = a_j\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0) + (0)$$
$$= \lambda a_j .\ \text{SINGLE}(a_i)(a_j) + \text{NULL}(a_j)$$
$$= \text{UNION}(\text{SINGLE}(a_i), \text{NULL}).$$

- UNION(s,s')

$$= \text{UNION}(\text{UNION}(\text{SINGLE}(a_{i_1}),\ldots,\text{UNION}(\text{SINGLE}(a_{i_m}),\text{NULL}))\ldots),$$
$$\text{UNION}(\text{SINGLE}(a_{j_1}),\ldots,\text{UNION}(\text{SINGLE}(a_{j_n}),\text{NULL})\ldots))$$
$$= \lambda a.((\underline{\text{if}}\ a=a_{i_1}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0)$$
$$+((\underline{\text{if}}\ a=a_{i_2}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0)$$
$$+\ldots$$
$$+(\underline{\text{if}}\ a=a_{i_m}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0) + 0))\ldots)$$
$$+ (\underline{\text{if}}\ a=a_{j_1}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0)$$
$$+\ldots$$
$$+ (\underline{\text{if}}\ a=a_{j_n}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0)+ 0))\ldots))$$
$$= \lambda a.(\underline{\text{if}}\ a=a_{k_1}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0)$$
$$+((\underline{\text{if}}\ a=a_{k_2}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0)$$
$$+\ldots$$
$$+((\underline{\text{if}}\ a=a_{k_{m+n}}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0) + 0))\ldots)$$
$$= \text{UNION}(\text{SINGLE}(a_{k_1}),\ldots, \text{UNION}(\text{SINGLE}(a_{k_{m+n}}), \text{NULL}))\ldots),$$

where $k_1 ,\ldots, k_{m+n}$ is the merge of $i_1 ,\ldots, i_m$ and $j_1 ,\ldots, j_n$.

The manipulations in this proof are lenghy but not at all difficult.

Correctness of Implementations

If D is any data type interchangeable with some final data type F ; then there is a Σ-homomorphism from D to F. Then, taking "correctness" and "interchangeability with the specification" to be synonymous, a complete (though certainly non-effective) test for correctness of an implementation is the existence of a homomorphism from the implementation to the final data type specification. By contrast the relationship obtaining between an algebraic specification and an implementation is indirect, requiring, in general, either an "equality interpretation" [10] or more axioms [16] to make the test complete.

The homomorphism $\underline{abs} : D \longrightarrow F$ assigns to each element of D the abstract element it represents. (Thus, it has been called an "abstraction function" [12].) It is not, perhaps, too surprising that, given a DS, $\underline{abs}$ is easily derived ; it is the map which assigns to each element the collection of all its values for the operations of the DS. More precisely, suppose the DS is given by :

$$\begin{aligned} DS(s_N) = \; & \{t_1[x]/x_1' \in s_1', \ldots, x_{n_1}' \in s_{n_1}'\} \\ & \cup \\ & \vdots \\ & \cup \{t_k[x]/x_1^k \in s_1^k, \ldots, x_{n_k}^k \in s_{n_k}^k\}, \end{aligned}$$

and that s_i is the sort of t_i , $1 \le i \le k$. Then $\underline{abs}$ is given by :

$$\begin{aligned} \underline{abs}_{D,s_n} &: D_{s_N} \longrightarrow F_{s_N} \\ &: d \longmapsto \langle \lambda d_1', \ldots, d_{n_1}' \,.\, \underline{abs}_{s_1} \left(t_1\left|\begin{matrix} d_1', \ldots, d_{n_1}' \\ x_1', \ldots, x_{n_1}' \end{matrix}\right.[d]\right), \\ & \qquad \vdots \\ & \qquad \lambda d_1^k, \ldots, d_{n_k}^k \,.\, \underline{abs}_{s_k} \left(t_k\left|\begin{matrix} d_1^k, \ldots, d_{n_k}^k \\ x_1^k, \ldots, x_{n_k}^k \end{matrix}\right.[d]\right)\rangle, \end{aligned}$$

and $\underline{abs}_{D,s_i}$ = identity function, for $s_i \neq s_N$.

For example, to prove M, with respect to the specification of MSet, we need to show that $\underline{abs}_{M,\mathrm{MSet}}$ is a homomorphism, where :

$$\begin{aligned} \underline{abs}_{M,\mathrm{MSet}} &: (M_{\mathrm{Atoms}})^* \longrightarrow (M_{\mathrm{Atoms}} \longrightarrow M_{\mathrm{Nat}}) \\ &: w \longrightarrow \lambda a : M_{\mathrm{Atoms}} \,.\, \mathrm{COUNT}_M(a,w). \end{aligned}$$

That is, each string maps to its COUNT function.

The homomorphism conditions are (calling the specified algebra S, suppressing occurrences of $\underline{abs}_{M,\mathrm{Atoms}}$ and $\underline{abs}_{M,\mathrm{Nat}}$, and omitting the subscripts from $\underline{abs}_{M,\mathrm{MSet}}$) :

- $\underline{abs}\ (NULL_M) = NULL_S$
- $\underline{abs}\ (SINGLE_M(a)) = SINGLE_S(a)$
- $\underline{abs}\ (UNION_M(w,w')) = UNION_S(\underline{abs}(w),\ \underline{abs}(w'))$
- $\underline{abs}\ (REMOVE_M(a,w)) = REMOVE_S(a,\ \underline{abs}(w))$
- $COUNT_M(a,w) = COUNT_S(a,\ \underline{abs}(w))$.

Because we derived abs from the DS, the last homomorphism condition is trivially true :

$$\begin{aligned} COUNT_M(a,w) &= (\lambda a' : M_{Atoms}.\ COUNT_M(a',w))(a) \\ &= COUNT_S\ (a, (\lambda a' : M_{Atoms}.\ COUNT_M(a',w))) \\ &= COUNT_S\ (a,\ \underline{abs}\ (w)). \end{aligned}$$

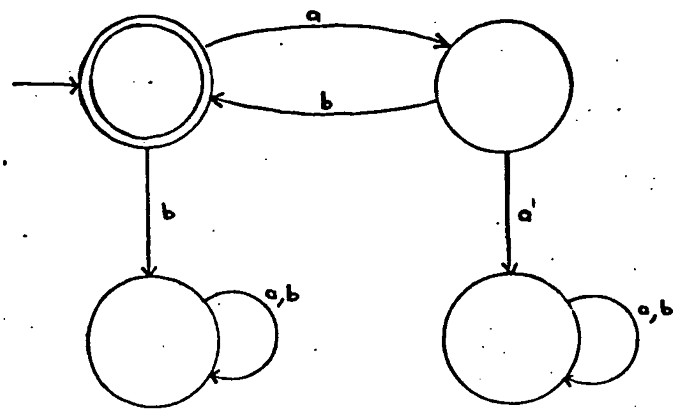
This will always hold for the functions appearing in the DS, when abs is derived in this way. There remain but four conditions to verify ; we give two of the proofs :

$$\begin{aligned} - \underline{abs}\ (NULL_M) &= \underline{abs}\ (\epsilon) \\ &= \lambda a : M_{Atoms}.\ COUNT_M(a,\epsilon) \\ &= \lambda a.0 \\ &= NULL_S \end{aligned}$$

$$\begin{aligned} - \underline{abs}\ (REMOVE_M(a,w)) & \\ &= \underline{abs}\ (\text{w with all a's removed}) \\ &= \lambda a'.\ COUNT_M(a',\ \text{w with a's removed}) \\ &= \lambda a'.\ \underline{if}\ a'=a\ \underline{then}\ 0\ \underline{else}\ COUNT_M(a',\ w) \\ &= \lambda a'.\ \underline{if}\ a'=a\ \underline{then}\ 0\ \underline{else}\ \underline{abs}\ (w)(a') \\ &= REMOVE_S(a,\ \underline{abs}\ (w)). \end{aligned}$$

The reader can readily fill in the two remaining proofs.

The finite-state automaton specification (Figure 2) requires induction on the structure of the domain. Suppose we wish to prove the following automat on correct :



Recall that the DS was {ACCEPT?(x)}∪{a(x)}∪{b(x)}, so the abstraction function is :

$$\underline{abs}_{States} : \{s_1, s_2, s_3, s_4\} \longrightarrow \text{infinite trees}$$
$$: s \longmapsto \langle ACCEPT?(s), \underline{abs}_{States}(a(s)), \underline{abs}_{States}(b(s))\rangle.$$

As mentioned, it is not necessary to verify the homomorphism conditions for ACCEPT?, a, and b. All we need to show is :

- $\underline{abs}_{States}(s_1)$ = START.

The proof is by fixpoint induction [20, 23] ; namely, we show that $\underline{abs}(s_1)$ satisfies the recursive definition of START :

$$\underline{abs}(s_1) = \langle true, \langle false, REJECT, \underline{abs}(s_1)\rangle, REJECT\rangle.$$

But

$$\underline{abs}(s_1) = \langle ACCEPT?(s_1), \underline{abs}(a(s_1)), \underline{abs}(b(s_1))\rangle$$
$$= \langle true, \underline{abs}(s_2), \underline{abs}(s_3)\rangle$$
$$= \langle true, \langle false, \underline{abs}(s_4), \underline{abs}(s_1)\rangle, \underline{abs}(s_3)\rangle.$$

So we need only show that $\underline{abs}(s_4) = \underline{abs}(s_3)$ = REJECT, which we do similarly, showing

$$\underline{abs}(s_4) = \langle false, \underline{abs}(s_4), \underline{abs}(s_4)\rangle.$$

Indeed, this is immediate, and the s_3 case is identical. (What this actually shows is that $\underline{abs}(s_1)$ is an extension of START, but since START has no proper extension, this gives $\underline{abs}(s_1)$ = START.)

Of course, our ability to do such proofs in general is limited by our knowledge of recursive functions. However, in practice, the proofs which arise are well within the power of a system such as LCF [8, 21].

V - CONCLUSIONS

Specifications of data types can usually be regarded as specifying particular algebras. Algebraic specifications always specify prime algebras (as long as "hidden operators" are eschewed). Final data type specifications always specify algebras whose prime part (i.e. smallest sub-algebra) is final. This situation is pictured in Figure 4.

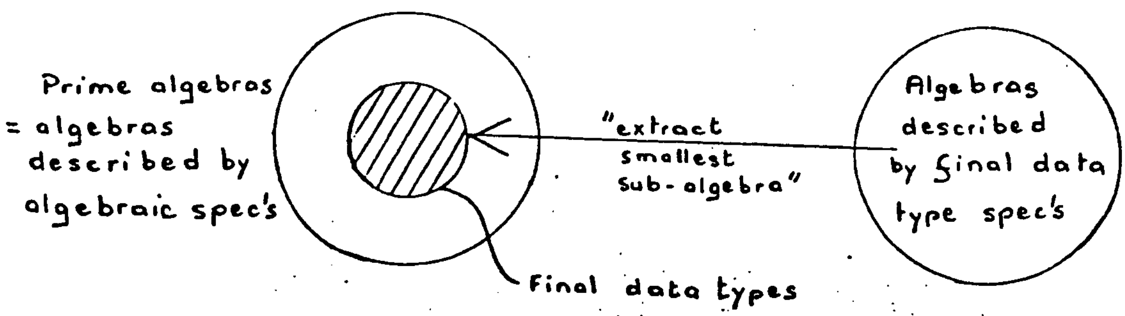


Figure 4

We have argued that the final data type is the most abstract representative of any given data abstraction. That is, the final data type contains the "essence" of the data abstraction, and, as such, has a particular mathematical relationship with other representatives of the data abstraction. (In denotational semantics, the analogous property of semantic domains is called "full abstraction" [22, 24].) By exploiting this relationship, we have given the first data type specification method which is - in a mathematical sense-entirely free of implementation bias [13, 19].

VI - APPENDIX

To define the method fully, we must say what solutions to domain equations of the form (*) are assumed. For purposes of explanation, let us consider the equation

$$A = B \times (C \longrightarrow A).$$

By a solution, we mean a complete partial order A with least element $\perp$, together with (continuous) functions $\Pi_1 : A \to B$ and $\Pi_2 : A \longrightarrow (C \longrightarrow A)$, and (continuous) bijection $<.,.>: B \times (C \longrightarrow A) \longrightarrow A$, satisfying the projection and tupling laws :

$$\forall b \in B \forall f \in (C \longrightarrow A).\ \Pi_1(<b,f>) = b \text{ and } \Pi_2(<b,f>) = f.$$

$$\forall a \in A. \qquad <\Pi_1(a), \Pi_2(a)> = a$$

To use final data type specifications, one need only know that such solutions exist, as follows from [18, 25]. To be more concrete, we give a construction ; assume B and C are cpo's with least elements, and define

$$A = C^* \to_\perp B,$$

where C^* = {sequences of elements of C, ordered componentwise, with sequences of differing lengths incomparable} $\cup$ $\{\perp_{C^*}\}$, and $C^* \to_\perp B$ is the set of continuous functions from C^* to B taking $\perp_{C^*}$ to $\perp_B$.

Then define

$$\begin{aligned}\Pi_1 &: (C^* \to_\perp B) \to B \\ &: a \mapsto a(\epsilon) \\ \Pi_2 &: (C^* \to_\perp B) \longrightarrow (C \to A) \\ &: a \mapsto \lambda c : C.\lambda c^* : C^*.a(c.c^*),\end{aligned}$$

where $\cdot : C \times C^* \to C^*$ is a concatenation operator which is strict in its second argument (but not its first). Finally,

$$\begin{aligned}<.,.> &: B \times (C \to A) \to (C^* \to_\perp B) \\ &: b,f \mapsto \lambda c^* : C^*.\ c^* = \perp \to \perp_B, \\ &\qquad\qquad c^* = \epsilon \to b, \\ &\qquad\qquad\qquad f(\mathrm{FIRST}(c^*))(\mathrm{REST}(c^*)).\end{aligned}$$

The reader is invited to verify the projection and tupling laws, and that <.,.> is a bijection. Also, this solution is minimal in that for any other solution D, there is an injective homomorphism (with respect to Π_1, Π_2, and <.,.>) from A into D. It is easily seen how this solution extends to equation (*).

Notice that every element of the space A is infinite. In particular, there is no distinction between $\perp_A$ and the element $\perp$ such that $\Pi_1(\Pi_2(\Pi_2(\ldots(\Pi_2(\perp)(c_1))(c_2))\ldots(c_n)) = \perp_B$ for any sequence $c_1c_2\ldots c_n \in C^*$.

This is required by our view that an element is characterized by its behavior, and is why we chose the equation we did in preference to, say, $A = (B \times (C \rightarrow A)) \cup \{\bot\}$.

REFERENCES

[1] M. Ardis, R. Hamlet,
"The structure of Specifications and Implementations of Data Abstractions",
University of Maryland, Computer Science Dept. TR-801, Sept., 1979.

[2] A. Bertoni, G. Mauri, P. Miglioli,
"Towards a Theory of Abstract Data Types : A Discussion on Problems and Tools",
International Symposium on Programming, Paris, April, 1980,
Springer-Verlag LNCS. 83, pp. 44-58.

[3] M. Broy, M. Wirsing,
"Abstract Data Types as Lattices of Finitely Generated Models"
9th Int'l Symp. on Foundations of Comp. Sci. , Lydzyna, Poland, 1980.

[4] J. Earley
"Toward an Understanding of Data Structures",
CACM 14, 1971, pp. 617-627.

[5] V. Giarratana, F. Gimona, U. Montanari,
"Observability Concepts in Abstract Data Type Specifications",
Math. Found. Comp. Sci., 76, pp. 576-587.

[6] J. Goguen, J. Thatcher, E. Wagner,
"An Initial Algebra Arrproach to the Specification, Correctness and Implementation of Abstract Data Types",
in Current Trends in Programming Methodology IV, R. Yeh (ed.),
Prentice-Hall, 1979, pp. 80-149.

[7] M. Gordon,
"The Denotational Semantics of Sequential Machines",
Info. Proc. Letters 10, 1, Feb., 1980, pp. 1-3.

[8] M. Gordon, R. Milner, R. Wadsworth,
Edinburgh LCF,
Lecture Notes in Comp. Sci. 78, Springer-Verlag, Berlin, 1979.

[9] J. Guttag,
"Abstract Data Types and the Development of Data Structures",
CACM 20, 6, June, 1977, pp. 396-404.

[10] J. Guttag, D. Musser, E. Horowitz,
"Abstract Data Types and Software Validation",
CACM 21, 1978, pp. 1048-1064.

[11] M. Harrison,
Introduction to Switching and Automata Theory,
Mc Graw-Hill, NY, 1965.

[12] C.A.R. Hoare,
"Proof of Correctness of Data Representations",
Acta Informatica 1, 1972, pp. 271-281.

[13] C.B. Jones,
"Implementation Bias in Constructive Specifications of Abstract Objects",
IBM Vienna, Sept., 1977.

[14] S. Kamin,
"Some Definitions for Algebraic Data Type Specifications",
SIGPLAN Notices 14, 3, March, 1979, pp. 28-37.

[15] S. Kamin,
"Final Data Type Specifications",
7th POPL Conference, Las Vegas, Nevada, 1980.

[16] S. Kamin,
"Specifications and Implementations".
Unpublished manuscript.

[17] P. Lescanne,
"Etude algébrique et relationelle des types abstraits et de leur représentation",
Thèse d'Etat, U. de Nancy, Sept.,1979.

[18] D. Lehmann, M. Smyth,
"Data Types",
Univ. of Warwick, England, Theory of Computation Report N° 19, 1977.

[19] B. Liskov, S. Zilles,
"Specification Techniques for Data Abstractions",
IEEE Trans. on Software Engg. SE-1, 1975, pp. 7-19.

[20] Z. Manna,
Mathematical Theory of Computation,
Mc Graw-Hill, NY, 1974.

[21] R. Milner,
"Models of LCF",
Stanford Memo AIM-186, Jan., 1973.

[22] R. Milner,
"Fully Abstract Models of Typed λ-calculus",
Theoretical Comp. Sci. 4, 1, Feb., 1977, pp. 1-22.

[23] D. Park,
"Fixpoint Induction and Proofs of Program Properties",
in B. Meltzer and D. Michie (eds),
Machine Intelligence 5, pp. 59-78, Edinburgh Univ. Press,
Edinburgh, 1969.

[24] G. Plotkin,
"LCF Condidered as a Programming Language",
Theor. Comp. Sci. 5, pp. 223-255.

[25] D. Scott,
"Data Types as Lattices",
SIAM J. Comput. 5, 1976, pp. 522-586.

[26] P. Subrahmanyam, R. Kieburtz,
"Toward Automatic Program Synthesis : Obtaining Implementations from
Formal Specifications",
SUNY at Stony Brook, Comp. Sci. Dept. TR-80, Sept., 1977.

[27] M. Wand,
"Final Algebra Semantics and Data Type Extensions",
JCSS 19, 1, Aug. 1979, pp. 27-44.