ISSN 0105-8517

DATA TYPES AS FUNCTIONS

by

Brian H. Mayoh

DAIMI PB-89

July 1978

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

Abstract

This paper introduces a new, simple definition of what a data type is. This definition gives ~~a precise meaning~~ one possible solution of the theoretical problems: when can ~~one~~ an actual parameter of type T be substituted for a formal parameter of type T'? when can a type T' be implemented as another type T'? The preprint is an extended version of a paper presented at MFCS 78, Zakopane.

DATA TYPES AS FUNCTIONS

This preprint is an extended version of [12]. The data types associated with equational and <u>cluster</u> specifications are described in more detail; the theory behind various techniques for verifying that one data type can be substituted for or represented as another is given in full.

1. THE PROBLEM

Currently there is much interest in the design of programming languages which allow:

(1) parametrized types, the construction of new types from old [1];

(2) operations that are polymorphic in that they have type parameters (e.g. an operation for sorting a vector of any size);

(3) limiting the operations that can be applied to a type [2];

(4) types that are abstract in that the type representation cannot be used outside the type declaration [3];

The first two of these raise the theoretical problem:

> When can an actual parameter of type T be substituted for a formal parameter of type T'?

This is similar to, but not the same as the theoretical problem for abstract data types:

> When can a type T be represented as another type T'?

In this paper we discuss these two theoretical problems, not the design of new programming languages. The literature on the design problem is extensive; those interested should begin by looking at such languages as CLU, ALPHARD, MESA, EUCLID [4] and pondering on the TINMAN requirements [5]. In the remainder of this section we give an example of

(1)-(4); in section 2 we emphasize the abstractness of all type declarations; in section 3 we give a simple, precise definition of data types and relate it to other definitions in the literature; in section 4 we propose a solution of the two theoretical problems.

The usual example of a parametrized type is STACK(EL) with stacks of values of type EL as its values. We can define polymorphic operations for such specifications as:

PUSH : STACK(EL) x EL → STACK(EL)

NEW : STACK(EL)

POP : STACK(EL) → STACK(EL)

TOP : STACK(EL) → EL

in many existing programming languages. In some of these languages we can limit the permissible operations on values of STACK(EL) to PUSH, NEW, POP and TOP. Once a limit has been put on the operations on values of STACK(EL), there may be many useful representations of this parametrized type.

The concept of a parametrized type should be distinguished from the construction of new types from types and constructors that are provided ab initio by the programming language. In PASCAL one can write

```
TYPE ENTRY = RECORD
                 identifier: ALFA;
                 attribute : INTEGER;
             END
```

and there are no parameters in this construction of a new type from the primitive types, ALFA and INTEGER. Suppose a language allows both the type ENTRY and the parametrized type STACK(EL) with operations PUSH, NEW, POP and TOP. Then the language may well allow the declaration

TYPE SymbolTable = STACK(STACK(ENTRY))

and the definition of operations for such specifications as

BlockEntry, BlockExit: SymbolTable → SymbolTable

Initialize: SymbolTable

Extend: SymbolTable x ENTRY → SymbolTable

Find, Offset, BlockNumber: SymbolTable x ENTRY → INTEGER.

Presumably the language will also allow one to limit the operations on values of SymbolTable to the seven specified above. If so, there may be many useful representations of the type SymbolTable - some using different representations of stacks, and some not even using stacks.

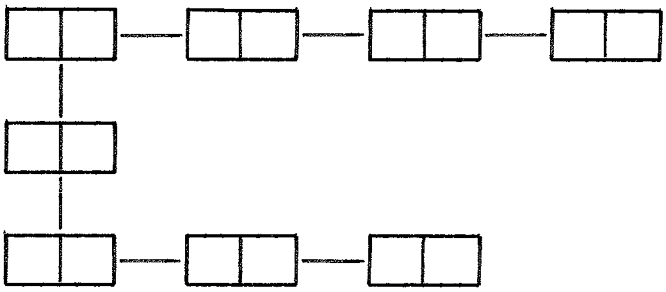


Figure 1. Values of the abstract type SymbolTable

2. ABSTRACT TYPES

All types are "abstract" in the sense that a programmer never knows what the values of a type really are. Suppose a language allows the definition of a type

VECTOR = ARRAY [low bound, high bound] OF REAL.

The programmer can read or write values into a variable v of type VECTOR by constructions like

```
        v [index] : = real value;
        some real variable : = ... v [index] ... ;
or      PUT (v, index, real value);
        some real variable : = ... GET (v, index) ... ;
```

The programmer may think of vectors as "tuples of reals", but she would never notice if vectors had been implemented in some quite different way satisfying

```
        PUT (v, index, GET (v, index)) = v ;
        GET (PUT (v, index, real value), index) = real value ;
```

Indeed there are good reasons for allowing different implementations of vectors – if the difference between the two bounds is large, but only a few vector components exist at any one time, an intelligent implementation might use a hashing function.

Although the programmer never knows what the values of a type really are, she usually knows how she wants to use them. If her program is to manipulate rational numbers, she should be able to introduce a data type with associated operations: ADD ... EQUALS. She should be able to write a declaration like

```
cluster
        RATIONAL = record
                        N, D : INTEGER;
                   end record;
        procedure Normalize (VAR x, y : INTEGER);
        (* code to replace the values of x, y by values such that x÷y*)
        (* is unchanged but x, y have no common divisor and y > 0   *)
within
```

```
function ADD (a, b: RATIONAL): RATIONAL;
    var c, d : INTEGER;
    begin
        c : = a.N * b.D + a.D * b.N; d : = a.D * d.D;
        Normalize (c, d);
        ADD.N : = c; ADD.D : = d;
    end function;
(* other public functions on rationals *)
function CREATE (a, b : INTEGER): RATIONAL;
    begin
        Normalize (a, b);
        CREATE.N : = a; CREATE.D : = b;
    end function;
function NUMERATOR(r: RATIONAL): INTEGER;
    NUMERATOR : = r.N;
function DENOMINATOR (r, RATIONAL): INTEGER;
    DENOMINATOR : = r.D;
function EQUALS (a, b : RATIONAL) : BOOLEAN;
    EQUALS : = (a.N = b.N) AND (a.D = b.D) ;
(* Note that our declaration format separates private and public      *)
(* operations, allows mutually recursive definitions or operations *)
(* and permits several types to be defined in the same  cluster       *)
end cluster
```

When the programmer writes a declaration, she reveals that she is thinking of rationals as "fractions in lowest terms as a pair of integers". Nevertheless the type RATIONAL is abstract in that there are many different declarations that give the same result for any computation on rationals that only uses the operations ADD ... EQUALS. An alternative declaration that avoids incessant renormalization is:

```
cluster
        RATIONAL = record
                           N,D : INTEGER;
                   end record;
        procedure  Normalize (VAR x,y : INTEGER);
        (*          as before                    *)
within
        function ADD (a,b : RATIONAL): RATIONAL;
            begin
                ADD.N : = a.N × b.D + a.D × b.N;
                ADD.D : = a.D × b.D;
            end function;
        (* other public functions on rationals *)
        function CREATE (a,b: INTEGER): RATIONAL;
            begin
                CREATE,N : = a; CREATE.D : = b;
            end function;
        function NUMERATOR (r: RATIONAL) : INTEGER;
            var a,b : INTEGER;
            begin
                a : = r.N; b : = r.D; Normalize (a,b);
                NUMERATOR : = a;
            end function;
```

```
function DENOMINATOR (r: RATIONAL):INTEGER;
    var a, b: INTEGER;
    begin
        a : r.N; b : = r.D; Normalize (a, b);
        DENOMINATOR : = b;
    end function;
function EQUALS (a, b: RATIONAL): BOOLEAN;
    EQUALS : = (a.N × b.D = a.D × b.N);
end cluster
```

We have given this rather detailed example to emphasize the theoretical problem – when can a type declaration be replaced by another type declaration without affecting the results of computations. To solve this problem we need a precise definition of what we mean by "type". This definition ought to give precise meanings to parametrized types and polymorphic operations. As both these concepts use types as parameters, this requirement on the definition of "type" can be rephrased: when can an actual parameter type be substituted for a formal parameter type. If our concept of parametrized type allows equations, we must be careful about the meaning of equality signs if we are to avoid the horrors of collapsing types (the authors of [6] were not and one can prove TRUE = FALSE from their definition of signed integer). Neither of our declarations of the type RATIONAL satisfies

$$\text{NUMERATOR}(\text{CREATE}(n, d)) = n$$

even although this equation seems natural.

3. DEFINITION AND SPECIFICATION OF DATA TYPES

What is a data type? One answer is: a computational rule that can be applied to expressions of the type to yield values of the type. In many cases we will have:

(1) values of the type can be stored, printed, passed as parameters ...;

(2) among the expressions of the type we have identifiers and a constant for each value;

(3) each value of the type is the result of applying the computational rule to some expression;

(4) applying the computational rule to an expression always yields a value;

but there is no reason to require these properties. However, there is no loss of generality if we insist on (4) in the form:

> the values of a type are partially ordered and there is a least element $\perp$ in this partial order

and thereby avoid partial function troubles.

Definition A data type is a total function f from a set of expressions E to a partially ordered set of values V with a least element in the partial order.

For us the data type problem is not to say what data types really are, but rather to single out the particular function associated with a program language text that purports to be a data type declaration.

Many data types are syntactic in the sense that their expressions are given by a grammar like

$$\langle \text{Boolean} \rangle ::= \text{TRUE} \mid \text{FALSE} \mid \text{NOT} \langle \text{Boolean} \rangle \mid \langle \text{Boolean} \rangle \text{ AND } \langle \text{Boolean} \rangle \mid \langle \text{Boolean} \rangle \text{ OR } \langle \text{Boolean} \rangle$$

Since Algol 60 the syntax of most programming languages has been given by such grammars so these languages can be regarded as syntactic data types. However, the fact that a data type is syntactic does not help in defining its computational rule, in giving the "semantics of the type". The best one can do if one has no information beyond the grammar is to define the computation rule $f: E \to V$ by $f(e) = \perp$ and take the singleton set consisting of $\perp$ as V. If one has information like:

> The set of values is $\{TT, FF, \perp\}$ and the computational rule is given by the usual truth tables

then the associated data type becomes more useful.

The simplest way of giving semantic content to a data type is by domain equation [7]. Scott and others have shown how to solve equations like

$$\text{StackR} \sim \{T\} + \text{RATIONAL} \times \text{StackR}$$

A solution of such an equation $Z \sim T(Z)$ consists of a particular set V and functions decode: $T(V) \to V$, project: $V \to T(V)$ such that decode $^\circ$ project is the identity on V. A solution gives a set of values for a type $f: E \to V$ and two functions that are useful in defining the computational rule, but one needs something else to define the set of expressions for a type. Our first declaration for the rationals gives a solution of the domain equation

$$\text{RATIONAL} \sim \text{INTEGER} \times \text{INTEGER}$$

with CREATE in the rôle of "decode" and NUMERATOR $\times$ DENOMINATOR in the rôle of "project". The solution does not have the identity as project $^\circ$ decode; we can find an integer pair $\langle n, d \rangle$ that is not the same as

$$\langle \text{NUMERATOR (CREATE}(n, d)), \text{DENOMINATOR(CREATE}(n, d)) \rangle$$

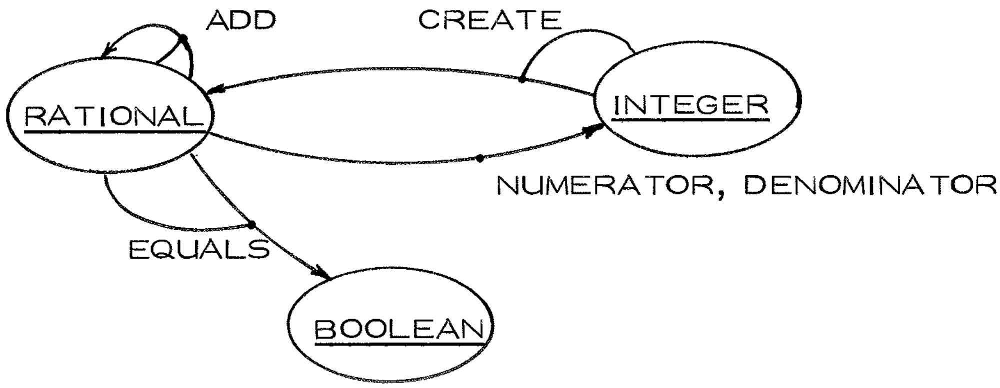
On the other hand we do have the equation

$$\text{CREATE (NUMERATOR}(r), \text{DENOMINATOR}(r)) = r$$

corresponding to the fact that decode $^\circ$ project is the identity.

To progress we could follow the tradition in programming language semantics (cf. syntax-driven compiler generators) and give names to the rules in a grammar. It seems more natural to follow the ADJ group [6] and introduce signatures instead of grammars. A signature Σ consists of a set SORT(Σ) and a set of operator symbols $\Sigma_{w,s}$ for each s in SORT(Σ) and each sequence w of elements from SORT(Σ). This sounds complicated but the following alternative representations of the signature for our data type RATIONAL should make the idea clear:

Picture representation:



Structure representation:

SORT(TRADITIONAL) = {RATIONAL, BOOLEAN, INTEGER}

ADD : RATIONAL × RATIONAL → RATIONAL

....

CREATE : INTEGER × INTEGER → RATIONAL

NUMERATOR, DENOMINATOR : RATIONAL → INTEGER

EQUALS : RATIONAL × RATIONAL → BOOLEAN

Grammar representation:

<RATIONAL>	::=	CREATE(<INTEGER>, <INTEGER>) |
		ADD(<RATIONAL>, <RATIONAL>) | ...
<INTEGER>	::=	NUMERATOR(<RATIONAL>) |
		DENOMINATOR(<RATIONAL>)
<BOOLEAN>	::=	EQUALS(<RATIONAL>, <RATIONAL>) |

The grammar representation shows that the data types given by a signature Σ are syntactic. If we have a set E_s for each s in SORT(Σ), then we can define $\Sigma[E_s]$ as the set of "words" built from elements of E_s by using the operator symbols in Σ. If we also have functions $\sigma_E : E_{s_1} \times E_{s_2} \times \ldots E_{s_n} \to E_{s_0}$ for each operator symbol σ in $\Sigma_{s_1 \, s_2 \, \ldots \, s_n, s_0}$ (in other words, if we have a Σ-algebra), we can define a function f by

$$f(e) = \underline{\text{if}}\ e \text{ is a form } \sigma[e_1, \ldots, e_n]$$
$$\underline{\text{then}}\ \sigma_E(f(e_1), \ldots, (f(e_n))\ \underline{\text{else}}\ e.$$

Now we have a data type f with $\Sigma[E_s]$ as its set of expressions and $\bigcup(E_s \mid s$ in SORT(Σ)) as its set of values. This data type f: $E \to V$ is rather special, it is a "retract", it satisfies : $V \subset E$, $f^\circ f = f$, f is the identity on V. An even more special type f_Σ is given by our construction when each E_s has only one element, the undefined element $\underline{\perp s}$ of sort s.

The two declarations of the type RATIONAL in the last section were disguised presentations of (ADD ... EQUALS)-algebra with carriers:

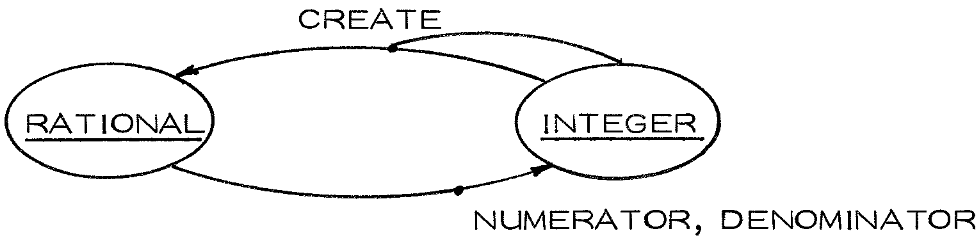
$E_{integer}$ = the usual integers,

$E_{boolean}$ = the usual booleans,

$E_{rational}$ = pairs of usual integers.

If the data type f: $E \to V$ given by our construction is to reflect the intended representation independence of the type declarations, the "words" built from $E_{rational}$ must be removed from the expression set E. The imprecision of the last two sentences highlights a crucial difficulty in the theory of abstract data types: we need to use predefined types when declaring new types, but these declarations may introduce new elements and undesirable properties in predefined types. As always, the most undesirable property is inconsistency and this can easily occur with equational specifications of types given be a signature.

Signature:



Equation set:

CREATE(NUMERATOR(r), DENOMINATOR(r)) = r

. . . .

Figure 2. Equational specification of a type

This is a convenient way of specifying the data type but the current agitation about error algebras [7], maximal equivalence classes [8], and final algebras [9] shows that it is not easy to say exactly what data type is being specified by a set of equations. Suppose we have an equational specification

$$L_i = R_i \quad \text{for } i \text{ in } I$$

of a type given by a signature Σ. If we have a set E_s for each s in SORT(Σ), we can define an equivalence relation $e \sim e'$ on $\Sigma[E_s]$:

> the equation $e = e'$ can be derived from the specification by substitution of elements of $\Sigma[E_s]$ for variables and the laws of equality.

Suppose we also have functions $\sigma_E : E_{s_1} \times E_{s_2} \times \ldots E_{s_n} \to E_{s_0}$ for each operator symbol σ in $\Sigma_{s_1 s_2 \ldots s_n, s_0}$. Suppose j is an interpretation, a map from variables to E_s. By substitution this map can be extended to words built from variables and operator symbols. The usual requirement for our Σ-algebra to be a model is

$$j(L_i) \text{ is the same element as } j(R_i)$$

for every interpretation j and every equation $L_i = R_i$.

Because $e \sim e'$ $\Rightarrow$ by substituting and then applying the laws of equality we can derive $e = e'$

the requirement gives

$$e \sim e' \Rightarrow f_E(e) \text{ is same element as } f_E(e').$$

We generalize this.

<u>Definition</u> A <u>model</u> of an equation specification with signature Σ consists of a set E_s for each sort s in SORT(Σ) and an equivalence relation $\equiv$ on $\Sigma[E_s]$ such that

(COMPATIBLE) $e \sim e' \Rightarrow e \equiv e'$

$\equiv$ is identity on E_s

The model is <u>operational</u> if we also have $e_1 \equiv e_1' \;\&\; \ldots \;\&\; e_n \equiv e_n' \Rightarrow \sigma[e_1, \ldots e_n] \equiv \sigma[e_1', \ldots e_n']$ for all operator symbols σ.

Any equational specification of a type given by a signature Σ has the model given by f_Σ - each E_s has just one element, and distinct elements are not equivalent. If the equational specification is to give new types from old, we must have E_s for some subset OLD of SORT(Σ). An element e of $\Sigma[E_s]$ is <u>determined</u> if its sort s is in OLD and there is an e' in E_s such that $e \sim e'$.

Any equivalence relation $\equiv$ satisfying (COMPATIBLE) agrees with $\sim$ on determined elements, and gives a candidate for <u>the</u> data type specified by the equations:

the set of expressions is $\Sigma[E_s]$

the set of values is the quotient set $\Sigma[E_s]/\equiv$

the computation rule is f(e) = equivalence class of e.

There are equivalence relations satisfying (COMPATIBLE) if and only if for every e in $\Sigma[E_s]$ there is at most one e' in E_s satisfying $e \sim e'$. The finest of these equivalence relations is $\sim$; the coarsest is $\div$ defined by: for any two elements of the same sort s we have $e_1 \div e_2$, unless there is no e' in E_s such that we have one of $e_1 \sim e'$ and $e_2 \sim e'$ but not both. The finest operational model is given by $\sim$, the model given by $\div$ may not be operational, the coarsest operational model is the final algebra of Wand [9].

Before we can explain parametrized types we must solve the theoretical problem

When can an actual parameter of type T be
substituted for a formal parameter of type T'?

A partial solution is: if all formal expressions of type T' occur in E then f: E → V can be substituted for T'; we give a complete solution in the next section.

Let us begin with polymorphic operations. Suppose we want to write a function LOOK to test if an element of an arbitrary type EL is in a vector v whose components are of type EL. Assuming "formal functions":

EQUALS : EL x EL → BOOLEAN

LowerBound, UpperBound : VEC → INTEGER

GET : VEC x INTEGER → EL

we can write the declaration:

```
function LOOK (v: VEC; e: EL): BOOLEAN;
var  i: INTEGER;
begin LOOK : = FALSE;
      for i : = LowerBound(v) to UpperBound(v)
      do if EQUALS(e,GET(v, i))
            then LOOK : = TRUE;
end function
```

Our declaration of the polymorphic operations LOOK does not convey semantic information directly. Assume that the formal expressions occurring in the declaration of a polymorphic operation LOOK belong to a "formal expression set E". The declaration of LOOK gives a data type, if we bind the parameters by giving some data type $f: E \to V$.

Consider our first cluster specification for the rationals. The functions ADD ... EQUALS are declared as polymorphic operations. Not only does the part of the specification between cluster and within determine

E_{INT} = values of type INTEGER

E_{BOO} = values of type BOOLEAN

E_{RAT} = pairs of values of type INTEGER,

but it also binds the parameters of the polymorphic operations - it gives values to formal expressions like

$$a.N \times b.D + a.D \times b.N$$

in the declarations of ADD ... EQUALS. The binding of the parameters gives functions

$$ADD'_E \; : \; E_{RAT}{}^2 \to E_{RAT}$$
$$\vdots$$
$$EQUALS'_E \; : \; E_{RAT}{}^2 \to E_{BOO}$$

and these give a data type $f': \Sigma[E_{RAT}, E_{INT}, E_{BOO}] \to E_{RAT}+E_{INT}+E_{BOO}$ by $f'(\sigma[e_1 \ldots e_m]) = \sigma'_E(f'(e_1, \ \ldots \ f'(e_m))$. If the declarations of

ADD ... EQUALS had been mutually recursive, we would have used a slightly different approach:

Given a function f'': $\Sigma[E_{RAT}, E_{INT}, E_{BOO}] \to E_{RAT}+E_{INT}+E_{BOO}$ the declarations give functions for each operator symbol, and these functions can be combined into a new function $F(f'')$: $\Sigma[E_{RAT}, E_{INT}, E_{BOO}] \to E_{RAT}+E_{INT}+E_{BOO}$. The function f' is the least fix point of F.

However we define f' we can extend ADD'_E ... $EQUALS'_E$ by

$$ADD_E(a,b) = ADD'_E(f'(a), f'(b))$$

....

$$EQUALS_E(a,b) = EQUALS'_E(f'(a), f'(b))$$

The data type f defined by the cluster specification is the restriction of f' to $\Sigma[\emptyset, E_{INT}, E_{BOO}]$ where $\emptyset$ is the empty set. The equivalence relation on $\Sigma[\emptyset, E_{INT}, E_{BOO}]$:

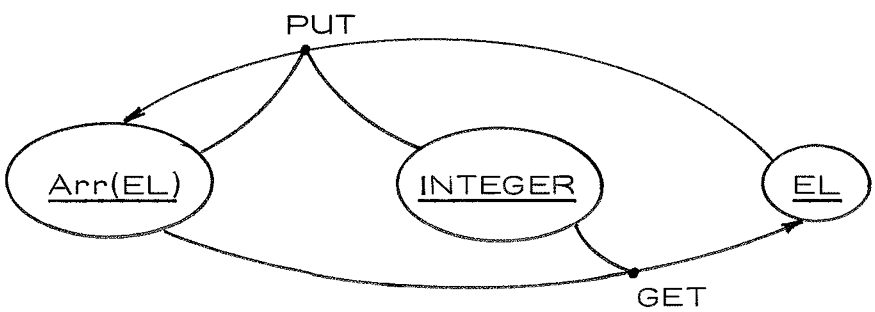
f(e) is the same element as f(e')

shows the connection with equational specifications.

Now for the declaration of parametrized types. Figure 3 shows an equational specification of the parametrized type UnboundedArray of (EL).

Signature

INIT



Equation set

GET (PUT(a, i, e) = e

for $i \neq j$ PUT (PUT(a, i, e), j, e') = PUT (PUT(a, j, e'), i, e)

Figure 3. Equational specification of a parametrized type

Suppose E_{INT} is the set of integer expressions and E_{EL} is the set of expressions of the type EL. Let $E_{\underline{ARR}(\underline{EL})}$ be the empty set so the set of expressions for the type specified by the equations are the words built from E_{INT} and E_{EL} using PUT, GET and INIT. We can show (COMPATIBLE) for the $\sim$ given by the equations in Figure 3. The only non-determined expressions are those of sort ARR(EL). All these expressions are $\div$ -equivalent whereas

(a) PUT(a, i, e_1) and PUT(a, i, e_2) are not equivalent in any operational model when e_1 and e_2 are different,

(b) PUT(PUT(a, i, e_1), i, e_2) and PUT(a, i, e_2) are equivalent in the coarsest operational model, but they are not $\sim$ - equivalent.

Since there is considerable debate about what is actually defined by an equational specification, it seems better to define this parametrized type from the functions decode and project given by a solution of the domain equation

$$Z \sim Z \times \text{INTEGER} \times \text{EL} + (T)$$

The functions for the operator symbols PUT and GET are given by

```
function PUT(a:ARR; i:INTEGER; e:EL):ARR;
        PUT := decode (a, i, e);
function GET(a:ARR; i:INTEGER):EL;
    case project (a) of
        <a', i', e> : if i = i' then e else GET(a', i)
    otherwise undefined;
```

The resulting data type can be different from that given by our equations in that there is no reason why decode should give the same value for the arguments PUT(PUT(a, 1, e_1), 2, e_2) and PUT(PUT(a, 2, e_2), 1, e_1).

This remains true even if the second equation is replaced by GET(PUT(a, i, e), j) = GET(a, j) for $i \neq j$ because project $\circ$ decode need not be the identity.

However we declare the parametrized type UnboundedArray of (EL), it can be used in such declarations as:

```
cluster
        STACK(EL) = record
                            contents: UnboundedArray of (EL);
                            pointer: INTEGER;
                        end record;
within
        function NEW : STACK;
             NEW.pointer : = 0;
        procedure PUSH (s: STACK, e : EL);
             begin PUT (s.contents, s.pointer, e);
                  s.pointer : = s.pointer + 1;
             end procedure;
        procedure POP (s: STACK);
             s.pointer : = s.pointer - 1;
        function TOP (s: STACK); EL;
             TOP : = GET (s.contents, s.pointer);
end cluster;
```

Here NEW, PUSH, POP and TOP are polymorphic operations with formal types STACK and EL. Like LOOK they lose their polymorphism when these formal types are bound to actual types. The actual type to be bound to the formal type STACK is given between record and end record. As soon as we have a set E_{EL} we get a set E_{arr} and a data type g that gives a meaning to the PUT and GET expressions in the cluster declaration. If we take $E_{arr} \times E_{integer}$ as E_{stack}, we get data types: NEW_g, $PUSH_g$, POP_g, TOP_g. These data types induce a data type for the cluster as a whole by:

$$f(\sigma[e_1 \ldots e_m]) = \sigma_g(f(e_1) \ldots f(e_m))$$

This data type should be compared with that given by the analogous construction for the declaration

```
cluster
    STACK(EL) = record
                    if not empty
                    then (head: EL; tail: STACK(EL))
                end record; (* see Figure 4 *)
within
    function NEW : STACK;
        NEW.not empty : = FALSE;
    function PUSH (s: STACK; e: EL): STACK;
        begin
            PUSH.not empty : = TRUE;
            PUSH.head : = e; PUSH.tail : = s;
        end function;
    function POP (s: STACK): STACK;
        POP : = if s.not empty then s.tail else undefined;
    function TOP (s: STACK): EL;
        TOP : = if s.not empty then s.head else undefined;
end cluster.
```

If we want to allow for errors and other stack operations, this declaration is easier to modify than the equivalent equational specification

$$\text{POP (PUSH}(s,e)) = s$$

$$\text{TOP (PUSH}(s,e)) = e$$

Our declaration used a recursive data type [10, 11, 1] for presenting a solution of the domain equation $Z \sim (T) + Z \times EL$; it can be reformulated using the equation functions decode and project instead of record field selection.

NEW	PUSH(NEW, e_0)	PUSH(PUSH(NEW, e_0), e_1)
= decode(T)	= decode(decode(T), e_0)	= decode(decode(decode(T), e_0), e_1))

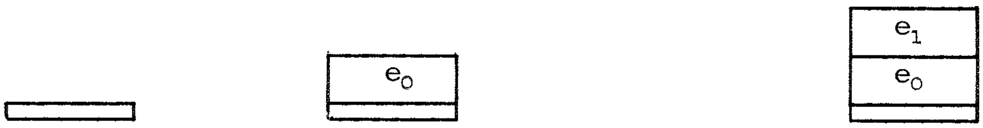


Figure 4. Value of Stack Type

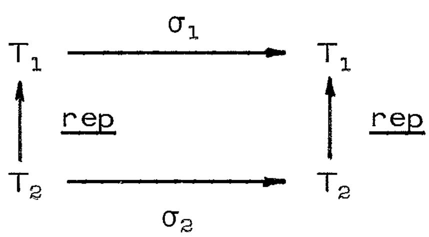
As another example of the use of recursive data types we give a cluster specification for unbounded arrays

```
cluster
        UnboundedArray of (EL) = record
                                      if not empty
                                      then (rest: UnboundedArray of (EL)
                                            index: INTEGER
                                            contents: EL)
        within
```

```
function PUT (a: ARR; i: INTEGER; e: EL): ARR
    begin
        PUT.not empty  : = FALSE;
        PUT.rest       : = a;
        PUT.index      : = i;
        PUT.contents   : = e;
    end
function GET (a:ARR; i: INTEGER): EL
    begin
        if a.index = i then a.contents
        else if a.not empty then GET(a.rest, i)
        else undefined
end cluster;
```

4. IMPLEMENTATION

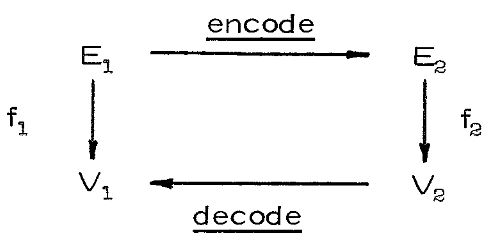
Now that we have a precise definition of a data type, we can tackle the problem – when can one data type T_1 be represented as another T_2 without affecting the results of computations. The first solution of this problem was given for another definition of data types [3]. The key idea is: operations on T_1 can be simulated by operations on T_2. For operations σ_1 from T_1 to T, simulation means the existence of functions rep and σ_2 such that the following diagram commutes



Analogous diagrams can be given for any operation on T_1.

The strong tendency to change the direction of rep arrows in such diagrams is a sign of the subtlety of the data type problem. Much discussion was provoked by an arrow reversal in a recent description [14] of how one might generate σ_2 automatically from rep and σ_1.

How can we use the idea of simulation, if we accept the definition of data type in the last section? The natural suggestion is to say that the data type $f_1 : E_1 \rightarrow V_1$ can be represented by $f_2 : E_2 \rightarrow V_2$ if there are functions encode and decode such that the following diagram commutes:



This is not quite adequate and becomes less so if one does not resist the temptation to reverse the bottom arrow (the counterpart of the rep arrow in the earlier diagram). The reason why it is not adequate is that it forces:

an undefined computation remains undefined
if one type is represented by another;

and this seems too stringent. It seems better to give a more general definition which reduces to the commuting of our diagram when f_1 is always defined and the partial order on V_1 is identity.

Definition A function encode: $E_1 \rightarrow E_2$ represents a data type $f_1 : E_1 \rightarrow V_1$ as a data type $f_2 : E_2 \rightarrow V_2$ if and only if there is a monotone function decode such that

(1) $\quad f_1 \subset \text{decode} \circ f_2 \circ \text{encode}$

A data type f_1 can be represented as a data type f_2 (abbreviated $f_1 << f_2$) if there is a function encode that represente f_1 as f_2.

Lemma << is transitive and reflexive.

Proof: Taking the identity for both encode and decode gives reflexivity. Suppose we have $\text{encode}_1 : E_1 \rightarrow E_2$, $\text{encode}_2 : E_2 \rightarrow E_3$, and monotone $\text{decode}_1 : V_2 \rightarrow V_1$, $\text{decode}_2 : V_3 \rightarrow V_2$ such that

$$f_1 \subset \text{decode}_1 \circ f_2 \circ \text{encode}_1$$

$$f_2 \subset \text{decode}_2 \circ f_3 \circ \text{encode}_2$$

From the second inclusion we get

$$f_2 \circ \text{encode}_1 \subset \text{decode}_2 \circ f_3 \circ \text{encode}_2 \circ \text{encode}_1 .$$

From the first inclusion and the monotonicity of decode_1 we get

$$f_1 \subset \text{decode}_1 \circ \text{decode}_2 \circ f_3 \circ \text{encode}_2 \circ \text{encode}_1 .$$

The transitivity of << is proven. □

Theorem Representations do not collapse types.

Proof: Suppose we have data types $f_1 : E_1 \rightarrow V_1$; $f_2 : E_2 \rightarrow V_2$, and functions decode and encode satisfying (1). Suppose e_1 and e_2 are elements of E_1 such that

$$f_1(e_1) \subset v \rightarrow f_1(e_1) = v$$

$$f_1(e_2) \subset v \rightarrow f_1(e_2) = v.$$

From (1) we infer

$$f_1(e_1) = \text{decode} \circ f_2 \circ \text{encode}\ (e_1)$$

$$f_1(e_2) = \text{decode} \circ f_2 \circ \text{encode}\ (e_2)$$

If we do not have $f_1(e_1) = f_1(e_2)$, we cannot have $f_2 \circ \text{encode}\ (e_1) = f_2 \circ \text{encode}\ (e_2)$ and our proof is complete. □

How can we represent a data type f_1 that is specified by a set of equations over a signature Σ? Suppose we have a set E_s for each s in SORT(Σ). If we also have a type $f_2 : E_2 \to V_2$ and a function <u>encode</u>: $\Sigma(E_s) \to E_2$, we can define an equivalence relation $\equiv$ on $\Sigma[E_s]$ by

$$e \equiv e' \text{ if and only if } f_2 \circ \underline{\text{encode}}\,(e) = f_2 \circ \underline{\text{encode}}\,(e').$$

In the case when $\equiv$ is a model of the equational specification, we can define <u>decode</u> by

$$\underline{\text{decode}}\,(v) = \text{the} \sim \text{equivalence class containing } e$$
$$\text{for all } e \text{ such that } f_2 \circ \underline{\text{encode}}\,(e) = v$$

and we have <u>encode</u> representing the data type f_1 specified by the equations as f_2. Suppose the type $f_2 : E_2 \to V_2$ is also specified by a set of equations over signature Σ. If we have

$e = e'$ can be derived from the first equation set

$\Rightarrow$ $e = e'$ can be derived from the second equation set

then any model of the second equation set will be a model of the first equation set. By the argument above identity represents the data type for the first equation set as the data type for the second equation set. These data types exist if the second equation set is consistent:

$e = e'$ can be derived for at most one e' in E_s.

<u>Lemma</u> For any data type $f: E \to V$ such that $f(E) = V$ there are data types $\hat{f}$ and $\underline{\overline{f}}$ such that

(a) $\hat{f}$ is a retract

(b) the value set of $\underline{\overline{f}}$ is the quotient of E by an equivalence relation

(c) $f_1 \ll f_2$ for f_1, f_2 in $< f, \hat{f}, \underline{\overline{f}} >$

Proof: Define the equivalence relation $\sim$ on E by

$$e \sim e' \Leftrightarrow f(e) = f(e')$$

Define $\bar{f}(e)$ as the equivalence class containing e. Pick representatives of the equivalence classes. Define $\hat{f}(e)$ as the representative of $\bar{f}(e)$. Clearly (a) and (b) are satisfied. Take the identity on E as encode. As f(E), $\hat{f}(E)$ and $\bar{f}(E)$ are order isomorphic we have decode for each choice of f_1 and f_2 in (c). □

Comment This lemma shows the connection between our definition of a data type, the Scott definition [7] and the algebraic definition [6]. The tedious requirement "f(E) = V" can be satisfied by "adding expressions to E" - this is most artificial when f is totally defined but we have to add an undefined expression because the partial ordering on V has a least element $\perp$.

Lemma Let $f_1 : E_1 \rightarrow V_1$ and $f_2 : E_2 \rightarrow V_2$ be data types. If E_1 is a subset of E_2 and f_1 is the always undefined function, then $f_1 << f_2$.

Proof: Take identity as encode and the constant function as decode. □

Comment In section 3 we defined a data type f_Σ for each signature Σ. Our lemma gives $f_\Sigma << f_E$ for any data type f_E given by a Σ-algebra. If E_S is both the carrier of such an algebra and the expression set of a type $f_S : E_S \rightarrow V_S$, then we have $f_S << f_E$ because we can take f_S as decode and identity as encode.

In the last section we gave a partial solution to the theoretical problem: When can an actual parameter of type T be substituted for a formal parameter of type T'? The similarity of the partial solution to the premiss of the last lemma suggests $T' << T$ as the complete solution. If we bind an actual parameter $f: E \to V$ to a formal parameter of type T', then the value of a formal expression e is $f \circ$ encode (e). Polymorphic operations like LOOK and parametrized types like STACK(EL) become data types, when formal expressions acquire values. Programming languages should only allow declarations of polymorphic operations and parametrized types that satisfy

(MONOTONICITY) $f_1 << f_2 \to G(f_1) << G(f_2)$

where G(f) is the data type given by the declaration for actual parameter f. Suppose G'(f) is the data type given by some other declaration for the same actual parameter f. If the two declarations accept the same actual parameters, and we have

(UNIFORMITY) $G(f) << G'(f)$ for all actual parameters f

we can prove such statements as

$G(G(f)) << G'(G'(ff))$.

In this way we can verify that such declarations as

SymbolTable = STACK(STACK(ENTRY))

do not depend on the way STACK and ENTRY are implemented.

5. VERIFICATION

In this section we describe three useful techniques for proving that a function encode: $E_1 \rightarrow E_2$ represents a data type $f_1: E_1 \rightarrow V_1$ as a data type $f_2: E_2 \rightarrow V_2$: structural induction, arrow reversal, generator induction. For most data types that occur in practice f_1, f_2 are least fix points of functionals $F_1: (E_1 \rightarrow V_1) \rightarrow (E_1 \rightarrow V_1)$, $F_2: (E_2 \rightarrow V_2) \rightarrow (E_2 \rightarrow V_2)$.

We say that f is the least fix point of the functional $F: (E \rightarrow V) \rightarrow (E \rightarrow V)$ if for every e in E we have

$$\forall n. F^n(\perp\!\!\!\perp)\, e \subset g(e) \Leftrightarrow f(e) \subset g(e)$$

where $\perp\!\!\!\perp$ is the always undefined function. When the partial order $\subset$ on V is such that directed chains have least upper bounds we can write this condition as

$$f(e) = \bigcup F^n(\perp\!\!\!\perp)\, e$$

and continuity gives $F(f)e = f(e)$ for all e. What do we mean by continuity here? The usual definition is in terms of partial orderings on sets like $E \rightarrow V$. In computing practice only the following special case arises

$$\text{(CONTINUITY)} \quad \forall e. \exists n. \quad F^n(\perp\!\!\!\perp)\, e = f(e)$$

and we can infer this when F has the property

$$g(e) = \perp \quad \text{or} \quad F(g)(e) = g(e)$$

for any e in E and any $g = F^n(\perp\!\!\!\perp)$.

Example

Let A be any Σ-algebra. The corresponding data type is the least fix point of the functional

$$F(g)(\sigma[e_1 \ldots e_m]) = \sigma_A(e_1) \ldots g(e_m)).$$

Assume the value of σ_A is undefined if any of its arguments is undefined. The value $F(\perp\!\!\!\perp)$ e is defined if and only if e has operator depth $< n+1$; it is the same as $F^n(\perp\!\!\!\perp)$ e if e has operator depth $< n$.

Suppose we have encode: $E_1 \to E_2$, decode $V_2 \to V_1$ such that

(1) F_1 (decode $\circ$ f $\circ$ encode) (e) $\subset$ decode $\circ$ F_2(f) $\circ$ encode (e)

(MONOTONICITY) $f(e) \subset f'(e) \Rightarrow F_1(f)\ (e) \subset F_1(f')\ (e)$

From $\perp = \perp\!\!\!\perp_1(e) \subset$ decode $\circ$ $\perp\!\!\!\perp_2$ $\circ$ encode (e) we can infer

$F_1^K(\perp\!\!\!\perp_1)\ e \subset F_1^K$(decode $\circ$ $\perp\!\!\!\perp_2$ $\circ$ encode) e

$\subset F_1^{K-1}$ (decode $\circ$ $F_2(\perp\!\!\!\perp_2)$ $\circ$ encode) (e)

.....

$\subset$ decode $\circ$ $F_2^K(\perp\!\!\!\perp_2)$ $\circ$ encode (e)

By (CONTINUITY) we have K such that

$$f_1(e) = F_1^K(\perp\!\!\!\perp_1)\ e$$

$$f_2(e) = F_2^K(\perp\!\!\!\perp_2)\ e$$

for any e in E_1. Substitution now gives

$$f_1 \subset \underline{\text{decode}} \circ f_2 \circ \underline{\text{encode}}$$

We have proved $f_1 << f_2$ by structural induction.

Example

Suppose A and B are Σ-algebras. The corresponding data types are the least fix points of the functionals

$$F_1(f)(\sigma[e_1 \ldots e_m]) = \sigma_A(f(e_1) \ldots f(e_m))$$

$$F_2(f)(\sigma[e_1 \ldots e_m]) = \sigma_B(f(e_1) \ldots f(e_m))$$

Requirement (1) becomes

$$\sigma_A(\underline{\text{decode}} \circ f \circ \underline{\text{encode}}(e_1) \ldots \underline{\text{decode}} \circ f \circ \underline{\text{encode}}(e_m))$$

$$\subset \underline{\text{decode}} \circ \sigma_B(f \circ \underline{\text{encode}}(e_1) \ldots f \circ \underline{\text{encode}}(e_m)).$$

We can infer this from

(2) $\sigma_A(\underline{\text{decode}}(b_1), \ldots \underline{\text{decode}}(b_m)) \subset \underline{\text{decode}} \circ \sigma_B(b_1 \ldots b_m)$

Although we can also infer (2) from

$$\sigma_A(\underline{\text{decode}}(b_1), \ldots \underline{\text{decode}}(b_m)) = \underline{\text{decode}} \circ \sigma_B(b_1 \ldots b_m)$$

It is usually much easier to prove (2) directly – to require decode to be a Σ-homomorphism is to require too much.

Example

In section 2 we had two declarations of types for rationals. Let $f_1 : E_1 \rightarrow V_1$ be the data type for the declaration inwhich rationals were "fractions in lowest terms", and $f_2 : E_2 \rightarrow V_2$ be the data type for the declaration that avoided incessant renormalization. Define decode by

decode (v) = if v is a rational then Normalize(v) else v.

Requirement (2) becomes

ADD_1 (Normalize(a), Normalize(b)) $\subset$ Normalize $\circ$ ADD_2 (a, b)

.....

$CREATE_1$ (a, b) $\subset$ Normalize $\circ$ $CREATE_2$ (a, b)

$NUMERATOR_1$ (Normalize(r)) $\subset$ $NUMERATOR_2$ (r)

$DENOMINATOR_1$ (Normalize(r)) $\subset$ $DENOMINATOR_2$ (r)

$EQUALS_1$ (Normalize(a), Normalize(b)) $\subset$ $EQUALS_2$ (a, b)

The cluster definitions give $V_1 = V_2 = \text{Integer}^2 + \text{Integer}$ where Integer = (... −1, 0, 1, 2 ...). If a' = Normalize(a) and b' = Normalize(b), there are integers k and l such that

$a.N = k \times a'.N$	$a.D = k \times a'.D$
$b.N = l \times b'.N$	$b.D = l \times b'.D$

Since Normalize(x, y) = Normalize(k × l × x, k × l × y) for all integers x, y, k, l we have

$$\text{Normalize}(a'.N \times b'.D + a'.D \times b'.N,\ a'.D \times b'.D)$$

$$= \text{Normalize}(a.N \times b.D + a.D \times b.N,\ a.D \times b.D)$$

and this is $\text{ADD}_1(\text{Normalize}(a), \text{Normalize}(b)) = \text{Normalize} \circ \text{ADD}_2(a, b)$. Because the representation of a rational in lowest terms is unique, we have

$$a'.N = b'.N \ \text{AND}\ a'.D = b'.D \Leftrightarrow a.N \times b.D = a.D \times b.N$$

and this is $\text{EQUALS}_1(\text{Normalize}(a), \text{Normalize}(b)) = \text{EQUALS}_2(a, b)$. The inclusions for CREATE, NUMERATOR, DENOMINATOR are

$$\text{Normalize}(a, b) \subset \text{Normalize}(a, b)$$

$$\text{Normalize}(r).N \subset \text{Normalize}(r).N$$

$$\text{Normalize}(r).D \subset \text{Normalize}(r).D$$

so they require no proof. We have verified that f_1 can be represented by f_2.

Our next verification technique is arrow reversal: to infer $f_1 << f_2$ from

(3) $\underline{\text{project}} \circ f_1 \subset f_2 \circ \underline{\text{encode}}$ and $\underline{\text{decode}} \circ \underline{\text{project}}(v) = v$.

Suppose we have $\underline{\text{encode}} : E_1 \rightarrow E_2$, $\underline{\text{project}}\ V_1 \rightarrow V_2$ such that

(4) $\underline{\text{project}} \circ f(e) \subset f' \circ \underline{\text{encode}}(e) \Rightarrow \underline{\text{project}} \circ F_1(f)(e) \subset F_2(f') \circ \underline{\text{encode}}(e)$

We can infer

$$\underline{\text{project}} \circ F_1^k(\perp\!\!\!\perp_1)\, e \subset F_2^k(\perp\!\!\!\perp_2) \circ \underline{\text{encode}}(e)$$

from $\quad \underline{\text{project}}(\perp) = \underline{\text{project}}(\perp\!\!\!\perp_1)\, e \subset \perp\!\!\!\perp_2 \circ \underline{\text{encode}}(e)$.

By (CONTINUITY) we have k such that

$$f_1(e) = F_1^k(\perp\!\!\!\perp)\, e$$

$$f_2(e) = F_2^k(\perp\!\!\!\perp)\, e$$

for any e in E_1. Substitution now gives

$$\underline{\text{project}} \circ f_1(e) \subset f_2 \circ \underline{\text{encode}}(e).$$

Example

Suppose A and B are Σ-algebras. Define F_1 and F_2 as before. Requirement (4) becomes

$$\underline{project} \circ f(e) \subset f' \circ \underline{encode}(\sigma[e_1 \ldots e_m])$$

$$\Rightarrow \underline{project} \circ \sigma_A(f(e_1) \ldots f(e_m)) \subset \sigma_B(f'(\underline{encode}(e_1) \ldots f'(\underline{encode}(e_m)).$$

This is satisfied when each σ_B is monotone and

$$\underline{project} \circ \sigma_A(f(e_1) \ldots f(e_m)) \subset \sigma_B(\underline{project} \circ f(e_1) \ldots \underline{project} \circ f(e_m))$$

for all f, σ, $e_1 \ldots e_m$. Thus we can infer (4) from

(5) $\underline{project} \circ \sigma_A(a_1 \ldots a_m) \subset \sigma_B(\underline{project}(a_1) \ldots \underline{project}(a_m))$

or the unnecessarily strong: project is a Σ-homomorphism.

Example

Once again let f_1 and f_2 be the data types for the two declarations of the rationals. Define project as the identity, and decode as before. We can verify $f_1 << f_2$ by proving decode $\circ$ project(r) = r and

$ADD_1 (a,b) \subset ADD_2 (a,b)$

....

$CREATE_1 \subset CREATE_2$, $NUMERATOR_1 \subset NUMERATOR_2$,

$DENOMINATOR_1 \subset DENOMINATOR_2$, $EQUALS_1 \subset EQUALS_2$

for the appropriate partial order on V_2.

We define the partial order $\subset$ on V_2 by

$v \subset v'$ <≡> v and v' are the same

OR v = Normalize(v')

We have to prove Normalize(r) = r and the inclusions

$$\text{Normalize}(a.N \times b.D + a.D \times b.N,\ a.D \times b.D)$$
$$\subset (a.N \times b.D + a.D \times b.N,\ a.D \times b.D)$$

$$\text{Normalize}(a', b') \subset (a', b')$$

r.N = Normalize(r).N

r.D = Normalize(r).D

(a.N = b.N) AND (a.D = b.D) = (a.N $\times$ b.D = a.D $\times$ b.N)

where a, b, r are normalized. As not all pairs of integers are in $f_1(E_1)$, this set of inclusions is easier to prove than our earlier set.

Our third verification technique is generator induction. If we define $\underline{\text{project}}^m$ by

$$\underline{\text{project}}^m(a_1 \ldots a_m) = <\underline{\text{project}}(a_1) \ldots \underline{\text{project}}(a_m)>$$

we can write requirement (5) as

(6) $\underline{\text{project}} \circ \sigma_A \subset \sigma_B \circ \underline{\text{project}}^m$,

and this is none other than (3) with $\underline{\text{project}}^m$ in the rôle of $\underline{\text{encode}}$.

If we define $\underline{\text{decode}}^m$ by

$$\underline{\text{decode}}^m(b_1 \ldots b_m) = <\underline{\text{decode}}(b_1) \ldots \underline{\text{decode}}(b_m)>$$

we can write requirement (2) as

(7) $\sigma_A \circ \underline{\text{decode}}^m \subset \underline{\text{decode}} \circ \sigma_B$.

Since $\underline{\text{decode}} \circ \underline{\text{project}}$ is the identity, this gives

$$\sigma_A \subset \underline{\text{decode}} \circ \sigma_B \circ \underline{\text{project}}^m .$$

The operations σ_A and σ_B are data types in their own right and we have $\sigma_A \subset\subset \sigma_B$ from either (6) or (7). Generator induction is our name for proving (6) or (7) by induction on the "structure of V_1".

<u>Example</u>

In the last section we had two declarations for STACK[EL]. Choose any data type as EL. Let $f_2 : E_2 \to V_2$ be the data type for the declaration in which stacks were unbounded arrays, and $f_1 : E_1 \to V_1$ be the data type for the other declaration. Define $\underline{\text{decode}}$: $V_2 \to V_1$, $\underline{\text{project}}$: $V_1 \to V_2$

and the partial order $\subset$ on V_2 by (see Fig. 5)

$$\underline{\text{project}}\ (\text{PUSH}(\ \ldots\ (\text{PUSH}(\text{NEW}, e_0), e_1)\ \ldots\ e_{p-1})$$

$$= \quad \langle a, p \rangle$$

$$\text{where GET}(a, i) = \underline{\text{if}}\ 0 \leq j < p\ \underline{\text{then}}\ e_p\ \underline{\text{else}}\ \perp$$

$$\underline{\text{project}}\ (\perp_{ST}) = \perp_{ARR}$$

$$\underline{\text{decode}}\ (a, p) = \text{PUSH}(\ldots\ \text{PUSH}(\text{NEW}, e_0), e_1)\ \ldots\ e_{p-1})$$

$$\text{where } e_i = \text{GET}(a, i)$$

$$\underline{\text{decode}}\ (\perp_{ARR}) = \perp_{ST}$$

$$\langle a, p \rangle \subset \langle a', p' \rangle \iff \langle a, p \rangle = \langle a', p' \rangle$$

$$\text{OR GET}(a, j) = \underline{\text{if}}\ 0 \leq j < p\ \underline{\text{then}}\ \text{GET}(a', j)\ \underline{\text{else}}\ \perp$$

Clearly we may not have $\underline{\text{project}} \circ \underline{\text{decode}}(a, p) = \langle a, p \rangle$, but we do have $\underline{\text{decode}} \circ \underline{\text{project}}(s) = s$ and $\underline{\text{project}} \circ \underline{\text{decode}}(a, p) \subset \langle a, p \rangle$.

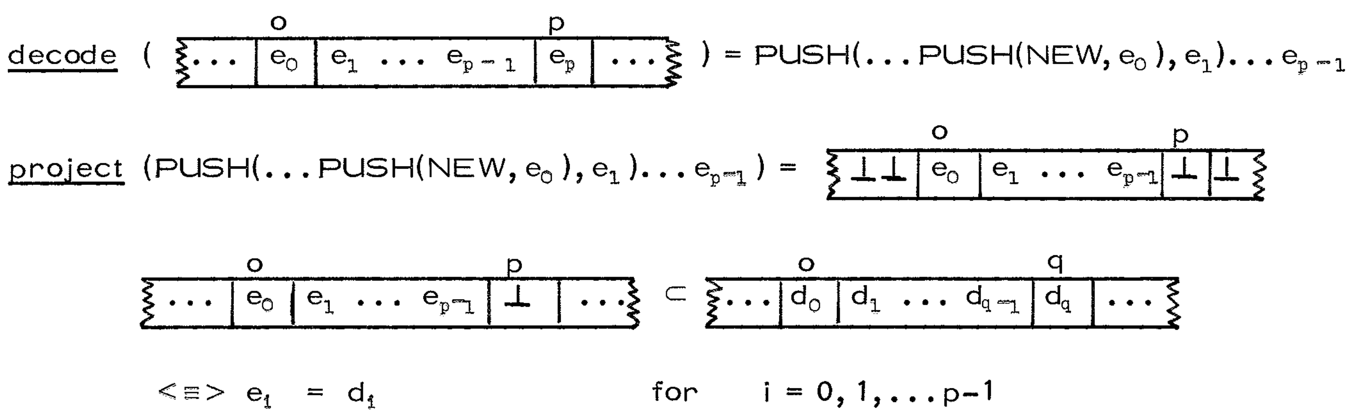


Fig. 5 Maps between stacks and unbounded arrays

To prove that the data type f_1 can be represented as the data type f_2, we can prove

$$\underline{\text{project}} \circ \text{NEW}_1 \subset \text{NEW}_2$$

$$\underline{\text{project}} \circ \text{PUSH}_1(s, e) \subset \text{PUSH}_2(\underline{\text{project}}(s), (e)$$

$$\underline{\text{project}} \circ \text{POP}_1(s) \quad \subset \text{POP}_2(\underline{\text{project}}(s))$$

$$\text{TOP}_1(s) \quad \subset \text{TOP}_2(\underline{\text{project}}(s))$$

Since project(NEW) = $< \bot, 0>$, we have the first inclusion from:

$< \bot, 0> \subset <a, p>$ for all a, p. Now suppose

s = PUSH(...PUSH(NEW, e_0), e_1)... e_{p-1}) and $<a, p>$ = project(s).

For $p > 0$ we have

$$\underline{\text{project}} \circ \text{PUSH}_1(s, e_p) = < a_1, p+1>$$

$$\underline{\text{project}} \circ \text{POP}_1(s) = < a_2, p-1>$$

$$\text{TOP}_1(s) = e_{p-1}$$

$$\text{PUSH}_2(<a, p>, e_p) = < \text{PUT}(a, p, e_p), p+1>$$

$$\text{POP}_2(a, p) = <a, p-1>$$

$$\text{TOP}_2(a, p) = \text{GET}(a, p-1)$$

where $\text{GET}(a, j) = \underline{\text{if}}\ 0 \le j < p\ \underline{\text{then}}\ e_j\ \underline{\text{else}}\ \bot$

$$\text{GET}(a_1, j) = \underline{\text{if}}\ 0 \le j < p+1\ \underline{\text{then}}\ e_j\ \underline{\text{else}}\ \bot$$

$$\text{GET}(a_2, j) = \underline{\text{if}}\ 0 \le j < p-1\ \underline{\text{then}}\ e_j\ \underline{\text{else}}\ \bot$$

Our inclusions become

$$e_j = \text{GET}(\text{PUT}(a, p, e_p), j) \text{ for } 0 \le j \le p$$

$$e_j = \text{GET}(a, j) \qquad \text{for } 0 \le j < p-1$$

$$e_{p-1} = \text{GET}(a, p-1).$$

In the case when s is NEW, or $\bot_{ST}$, the inclusions are also satisfied because project $\circ$ $\text{POP}_1(s)$ and $\text{TOP}_1(s)$ are undefined.

Since we can formulate the proofs of these inclusions without information about EL, they ensure that stacks can indeed be represented by unbounded arrays.

REFERENCES

[1] C.A.R. Hoare: Notes on data structuring, in [13].

[2] O.J. Dahl, C.A.R. Hoare: Hierarchical program structures, in [13].

[3] C.A.R. Hoare: Proof of correctness of data representations, Acta Informatica 1 (1972) 271-281.

[4] CLU, MESA, ALPHARD: see Comm. ACM 20 (1977) no. 8.
EUCLID : SIGPLAN Notices 12 (1977) 1-79.

[5] DOD requirements for high order computer programming languages, Springer Lecture Notes in Computer Science 54 (1977) 446-496.

[6] J.A. Goguen, J.W. Thatcher, E.G. Wagner: An initial algebra approach to the specification, correctness, and implementation of abstract data types, IBM Research Report RC6487 (1977).

[7] D. Scott: Data types as lattices, SIAM J. Comp. 5 (1976) 522-587.

[8] J.A. Goguen: Abstract errors for abstract data types. IFIP Conf. on Formal Description of Programming Concepts (1977) 21.1-21.32.

[9] J. Guttag, E. Horowitz, D.R. Mosser: Abstract data types and software validation. USC Inf. Sci. Inst. Report RR-76-48 (1976).

[10] M. Ward: Final Algebra semantics and data type extensions, Indiana Univ. Tech. Report 65 (1977).

[11] C.A.R. Hoare: Recursive data structures, Int. J. Comp. Inf. Sci. 4 (1975) 105-132.

[12] Short version of this preprint, to appear in Springer Lecture Notes in Computer Science, Proc. MFCS 78.

[13] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: Structured Programming, Academic Press 1972.

[14] J. Darlington: Program transformation involving unfree data structures, 3eme Coll. Int. sur la Programmation (1978) Paris.