DATA TYPES AS TERM ALGEBRAS

by

Akira Kanda and Karl Abrahamson

Technical Report 83-2

March 1983

Data Types as Term Algebras

Akira Kanda   Karl Abrahamson
Department of Computer Science
University of British Columbia
Vancouver, B.C.
Canada

Abstract

Data types in programming have been mathematically studied from two different viewpoints, namely data types as (initial) algebras and data types as complete partial orders. In this paper, we explore a possibility of finitaristic approach. For finitarists, the only sets accepted are "recursively defined" sets. We observe that recursive definition not only defines a set of terms but also basic operations over them, thus it induces an algebra of terms. We compare this approach to the existing two approaches. Using our approach we present finer classification of data types.

§1. Term Algebras vs. Initial Algebras

A view of data types as initial algebras, developed by ADJ group [1,2], has been widely accepted and some programming systems based on this idea have been developed (see Goguen [3]). In this approach, we specify a data type as a collection Σ of many sorted operation symbols (called signature) and a collection E of equational axioms which operations in Σ have to satisfy. To guarantee the uniqueness of the specification (Σ,E), we choose the initial algebra $I_{(\Sigma,E)}$ which satisfies E as the specified data type. How to construct $I_{(\Sigma,E)}$ from (Σ,E), can be found in ADJ [1,2].

A good example of initial algebra specification method is as follows:

Σ = {zero:->int, suc:int->int, pred:int->int}

E = {suc(pred(x)) = x, pred(suc(x)) = x}.

The data type specified is an initial algebra $I_{(\Sigma,E)}$ which interprets Σ as follows:

int |--> the set Z of all integers

zero |--> the constant $0 \in Z$

suc |--> λx.x+1:Z->Z

pred |--> λx.x-1:Z->Z.

This specification method is based on the following philosophy:

"Carrier sets (sets obtained as the interpretation of sorts) of a data type should be induced from the equational property of operations."

In this paper, we agree that data types are algebras but we present an alternative specification method based on the following alternative philosophy of finitarists:

"Recursive structure of carrier sets of a data type should induce basic operations and the property of them."

In order to explain this approach, we present some examples of data type specifications.

Data type of natural numbers

--We start with "recursive definition" of natural numbers:

(Base): 0 is a natural number.

(Step): if n is a natural number then so is s(n)

(Closure): nothing is a natural number unless it is proved to be a natural number using (Base) and (Step).

This definition gives us a set of terms (over $\{0,s,(,)\}$).

--Now we induce operations of natural numbers from the "recursive definition" given above.

Notice that (Closure) ensures:

"if n is a natural number then
either it is 0
or it is s(m) for some unique natural number m."

This property gives us a predicate is-zero on terms s.t.

is-zero(n) = true if n=0
false otherwise.

(Step) shows how to compose and decompose on terms. This gives rise to the following two operations on terms:

suc(n) = s(n)

pred(n) = m if n = s(m)
undefined otherwise.

Thus we obtained an algebra over the terms of natural numbers.

Generally speaking, all term algebras derived from "recursive definition" of carrier sets contain predicates like "is-zero". Even though we do not make explicit reference, we assume term algebras have Boolean sort {true, false} as one of their carriers. Also notice that we are bound to obtain "partial" operations like "pred", thus partial algebras. Conceptually, this causes no problem as long as we agree that undefinedness propagates throughout expressions, for then we know exactly when an expression is undefined. We can make all operations of term algebras total by throwing in "undefined" value to each sort and making undefinedness propagation explicit. Then the data type of natural numbers will be "totalized" as follows:

is-zero(n) = true if n = 0
undefined if n = undefined
false otherwise

suc(n) = s(n) if n $\neq$ undefined
undefined otherwise

pred(n) = m if n = s(m)
undefined otherwise.

Term algebras canonically obtained from "recursive definition", as above, is called a recursive term algebra.

Let us compare this specification with the initial algebra specification. It is quite obvious that the process of generating the set of natural numbers from the recursive definition is the same as generating a free algebra from a constant 0 and a unary function s. In fact the initial algebra $I_{(\Sigma,E)}$ where

$\Sigma = \{0{:}\text{->}\underline{nat},\ s{:}\underline{nat}\text{->}\underline{nat}\}$

E = the empty set

gives us the set of natural numbers {0,s(0),s(s(0)),...} and the successor function suc as the interpretation of nat and s respectively. But $I_{(\Sigma,E)}$ will not give us any extra operation which is not in Σ, therefore we can not obtain predecessor function, which is associated with the free generation of the set of natural numbers.

To include predecessor operation in the initial algebra specification, we have to start with a signature $\Sigma' = \{$0:->nat, s:nat->nat, p:nat->nat$\}$ which includes a symbol p for the predecessor operation, then we take the initial algebra $I_{(\Sigma',\emptyset)}$ which is the free algebra generated from Σ'. But this algebra is no good because we have too many terms representing the same natural number. For example all of s(0), s(p(s(0))), p(s(s(0))) denote 1. Therefore we need a set of equational axioms which enforces different terms denoting the same object to be equal. Since initial algebra method works only for "total algebras", we have to totalize the predecessor operation with the aid of undefined elements. This makes the set of axioms quite elaborate. In fact the following set E' of axioms works:

```
p(0) = undefined
OK(0) = true
OK(s(n)) = OK(n)
OK(undefined) = false
p(undefined) = undefined
s(undefined) = undefined
IFE(true,n,m) = n
IFE(false,n,m) = m
IFE(undefined,n,m) = undefined
IFE(OK(n),P(s(n)),undefined)=IFE(OK(n),n, undefined)
```

IFE(OK(n),s(P(s(n))),undefined)=IFE(OK(n),s(n),undefined)

For the detailed explanation of how this works, readers can safely be referred to ADJ [2]. But we can conclude that the initial algebra which satisfies E' interprets nat,p,s to be the set of natural numbers, predecessor function, successor function respectively.

It should be noticed that E' is an equational characterization of the following properties:

(1) pred(0) is undefined.

(2) undefinedness propagates

(3) pred(suc(n)) = n

(4) if n $\neq$ 0 then suc(pred(n)) = n.

In fact when we define the data type of natural numbers using term algebra method, all of these properties can be proved by the structural induction on the structure of natural numbers, which is due to the recursive definition. For example (4) can be proved as follows:

(Induction Base) n = 0 implies n $\neq$ 0 is false. Thus (4) holds.

(Induction Step) Assume n $\neq$ 0 implies suc(pred(n)) = n

suc(pred(suc(n))) = suc(n) for suc(n) $\neq$ 0.

Thus for all natural number n, n $\neq$ 0 implies suc(pred(n)) = n.

In general, axiomatic specification is not straightforward. In history, quite a number of axiomatic systems proposed turned out to be inconsistent. As ADJ [2] pointed out, even equational theories are not exceptions. Quite often, avoiding inconsistency would lead us to somewhat unnatural specification, like E'.

In initial algebra specification, one ought to make sure not only the consistency but also if what he specified is what he wanted. For this purpose, he has to know the construction of the initial algebra $I_{(\Sigma,E)}$ from (Σ,E). This is not very easy for non-mathematicians. In case $E \neq \emptyset$, $I_{(\Sigma,E)}$ is the quotient of the free algebra generated from Σ, by the minimal congruence relation containing E. Therefore we have to work on an algebra over equivalence classes. Goguen et al. [2] showed how to obtain an algebra of representatives from the quotient algebra. But this choice of representatives is not quite natural.

In initial algebra specification, a set of specification axioms gives us a ground for reasoning about the data type specified. Also initiality gives us an induction method for reasoning, as shown in Goguen & Meseguer [4]. In our approach, we reason about data types using structural induction on the well-founded structure of terms.

These observations indicate that term algebra method is worth pursuing. It looks conceptually easier than initial algebra method. Especially treatment of undefinedness seems simpler in term algebra method than in initial algebra method.

The next thing to question is how far can we go with this approach. Can we specify all data types which initial algebra method can? In the rest of this paper, we will examine this question. To begin with, we present a recursive term algebra specification of the data type of integers.

Data Type of Integers

--"Recursive definition" of integers:

1. s(0) is a positive number.

2. If n is a positive number then so is s(n).

3. 0 is an integer.

4. if n is a positive integer then both

    n and -n

   are integers.

5. nothing is an integer unless it is proved to be so from 1 to 4.

From 5, we can conclude:

"if x is an integer then either

$x = 0$,

or $x = s(0)$,

or $x = s(n)$ for some positive number n,

or $x = -n$ for some positive number n."

This gives us the following predicates:

```
is-zero(x) = true    if x = 0
             false otherwise
is-one(x) = true     if x = s(0)
            false  otherwise
is-greater-than-one(x)  =  true  if  x  =  s(n)  for  some
                                      positive n
                          false otherwise
is-negative(x) = true if x = -n for some positive n
                 false otherwise
```

From 1, 2 and 4, as composition and decomposition operators, we have the following basic operations:

suc'(n) = s(n) if n = 0

```
          undefined otherwise
suc"(n) = s(n) if n is a positive number
          undefined otherwise
pred'(n) = 0 if n = s(0)
           undefined otherwise
pred"(n) = m if n = s(m) for some positive m
           undefined otherwise
neg'(n) = -n if n is a positive number
          undefined otherwise
neg"(m) = n if m = -n
          undefined otherwise
```

Even though we do not make it explicit, we can make the term algebra "total", as we did for natural numbers.

This finitarist version of integers can very easily be related to algebraist version of them as follows:

--From the basic predicates, we can define usual predicates:

```
is-zero     = is-zero
is-negative = is-negative
is-positive(n) = is-one(n) v is-greater-than-one(n)
```

--From basic operations, we can define usual operations:

```
suc(n) = suc'(n) if n = 0
         suc"(n) if n is a positive number
          0       if n = -s(0)
         neg"(pred"(m)) if n = -m and m ≠ s(0)
neg(n) = neg'(n) if n is a positive number
          0       if n = 0
         neg"(m) if n = -m
```

```
pred(n) = pred'(n) if n = s(0)
          pred"(n) if n = s(m) for some positive m
          -s(0)     if n = 0
          neg'(suc"(m)) if n = -m
```

Furthermore, by structural induction, we can easily prove:

```
pred(suc(x)) = x
suc(pred(x)) = x.
```

In term algebra specification, we have to make sure that a "recursive definition" of terms provide a unique representation for each element of the intended set. Then we automatically get sufficient collection of operations. On the contrary, in initial algebra specification, we have to make sure that we have chosen enough operations for the data type to be specified. Then by equational axioms, we enforce different terms denoting the same object to form an equivalence class. We think, even though it sometimes gets a bit tricky as above, providing unique representation via "recursive definition" is not conceptually difficult.

It is well known that there are some structures for each element of which it is impossible to provide a unique representation. Finite power set is such an example. How to handle them will be discussed later.

In summary, our motto is "data types as recursive term algebras." In the following, we will examine the relevance of this motto and propose a notion of data types based on this motto.

§2. Definable Operations (Computability)

One of the advantages of recursive term algebras is that they have "computational completeness". More specifically, from basic operations of term algebras, by iterative use of:

- function composition
- conditional definition (definition by cases)
- general recursive definition,

we can recursively define all partial computable functions over the terms recursively defined. For this reason we call term algebras complete data types. For example, it is well-known that partial recursive functions are exactly recursively definable functions of the data type of natural numbers.

Recursive term algebras provide an interesting sub-class of definable operations. We say an operation of a recursive term algebra is structurally recursive if it can be defined from basic operations by iterative use of:

- functional composition
- conditional definition
- structural recursion,

For example, all primitive recursive functions are structurally recursive functions over the data type of natural numbers. Conceptually they are the same except undefined value. In recursive function theory pred(0) = 0. Thus those structurally recursive functions which have undefined in their ranges are not primitive recursive functions. But all structurally recursive functions terminate and we can prove properties of them by structural induction. It should be noted that Klaeren [5] observed the importance of decomposition of objects in algebras. He discussed structural recursion and structural induction on decomposable algebras, i.e. algebras which permit unique well-founded decomposition of each element. It is obvious that structural recursion on recursive term algebras is a special case of Klaeren's general argument. The point we are making here is that from a computational point of view we might not need to get into the full generality of Klaeren. It seems as though structural recursion makes computational sense only when applied to recursive term algebras. At least it is nicer to be able to talk about structurally recursive functions within the theory of partial computable functions.

Even a smaller subclass of definable operations is interesting. We say an operation of a recursive term algebra is <u>primitively definable</u> if it is defined from basic operations using

- functional composition
- definition by cases.

Obviously all primitively definable operations are structurally definable and all structurally definable operations are recursively definable.

In summary, we observe that a recursive term algebra provides a basis for the computability over them.

§3. Functional Data Types (Do we miss them?)

It is well known among finitarists that whenever we define a set "recursively" we can decide if two elements of the set are equal or not. This fact is algebraically realized by the fact that we can define a total computable (in fact structurally recursive) predicate is-equal which satisfies the following property:

$$\text{is-equal}(x,y) = \begin{cases} \underline{\text{true}} \text{ if } x = y \\ \underline{\text{false}} \text{ otherwise.} \end{cases}$$

This indicates that we can not define functional data types by term algebra specification method.

We observe that even though functionality of functions is important for compile time type checking and efficient code generation, data types of functions are not useful for programming. In PASCAL [6], we specify functionality of each function, but functions of the same functionality do not form a data type. Functionality is used mainly for compile time type checking and efficient code generation. In ALGOL 68[7], we can define a mode of functions with the same functionality, but these modes are under much stronger constraint than non-functional modes. For example, recursive mode specification using functional mode is not allowed in ALGOL 68 (see Peck [8]). It seems that we can not make much use of function types except providing functionality information to compilers to assist compile time type checking.

The following observation on "qualitive" difference between the role in computing of non-functional data object and that of functional data object will explain why functional data should be treated differently from non-functional data. Let us consider the following computation of LISP:

```
rev(x):= if atom(x) then x else cons(rev(cdr(x)),rev(car(x )).
```

A LISP data object x (which is a binary tree) is decomposed into cdr(x) and car(x) to split the computation rev(x) into subcomputations rev(cdr(x)) and rev(car(x)). Also think of the following computation of natural numbers:

```
f(x):= if x = 0 then 1 else x*f(x-1).
```

A natural number x is decomposed into x-1 to obtain a subcomputation f(x-1). In the following computation:

```
μ(x):= f(0)
f(y):= if R(y) then y else f(y+1)
```

we start with 0 and keep composing by y+1 until R(y) becomes true. This function is known as $\mu y.R(y)$ in recursive function theory. In fact, iteratively decomposing and composing data object is the heart of computation on non-functional data object.

When it comes to functions, there is no obvious way of structuring them to allow composition and decomposition. Scott [9] proposed structuring them by extensional ordering $\sqsubseteq$ and decompose a function f into a chain $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ s.t. the least upper bound $\sqcup f_i$ of it is f. Given a computable function f, there is no canonical method of decomposing f, but given a procedure to compute f we can canonically decompose f. For example

```
f(n):= if n = 0 then 1 else n*f(n-1)
```

can be decomposed as follows:

Let $F = \lambda f.\lambda n.$ if n = 0 then 1 else n*f(n-1) and

$\bot$ = everywhere undefined procedure.

Then:

```
$f_0 = \bot$
$f_{i+1} := F(f_i) = \lambda n.$ if n = 0 then 1
                                 else if n = 1 then 1
                                 else if n = 2 then 1*1*2

                                 else if n = i then 1*1*2*--*i
                                 else $\bot$
```

form a chain $f_0 \sqsubseteq f_1 \sqsubseteq \ldots$ s.t. $f = \bigsqcup f_i$. This idea of structuring computable functions of the same functionality provides an excellent ground for understanding and reasoning functional computation mathematically. But this is not a common way how programmers understand functional computation. For example, when we write a program to functionally compose two functions f and g, we do not decompose f and g as $\{f_i\}$ and $\{g_i\}$ and then form a chain $\{q_i.f_i\}$ and take the limit $\bigsqcup g_i.t_i$ to get g.f. Most programmers will write the following function declaration for this:

Compose(f,g,x):= g(f(x))

When we pass functions f' and g' to Compose, they are substituted for f and g in g(f(x)). This indicates that, in real life programming, functions are just texts and operation over them is just substitution. This is also the view of logicians over functions. Church [10] called this a view of functions as intensions. For example, in λ-calculus, syntactic substitution is what β-reduction rule is all about. Also Gődel [11] took the same view in his theory of primitive recursive functionals of finite types. The understanding of functions which Scott took is what logicians call functions as extensions.

Anyway the role of functions as data under the intensional view (which is also commonly accepted view in programming) is not quite active compared with non-functional data. This suggests that we would not get very much from including function spaces in "user definable" data types as programming facility. We claim that the function spaces are for reasoning about programs extensionally rather than for programming feature. We have to notice that using function space is not the only way to reason about programs. There are quite a few ways to do it intentionally (see Gilmore [12] for example). The following statement will explain our viewpoint very clearly:

"Functions of the same functionality form not a data type but a type".

§4. Finite Data Types

"Recursive definition" of sets allows us to define finite sets of terms. For example:

1. t_1 is foo
2. t_2 is foo
3. t_3 is foo
4. nothing is a foo unless it is proved to be foo from 1, 2, 3.

is a "recursive definition" and the set of foos defined is $\{t_1,t_2,t_3\}$. Basic operations associated with this "recursive definition" are:

$is\text{-}t_i(x) = \underline{true}$ if $x = t_i$ $(1 \le i \le 3)$.

It is easy to observe that all computable functions over a finite set of terms are primitively definable from basic operations is-t_i. It is needless to say that equality check is a computable function.

§5. Hidden Functions

Returning back to the specification of integers, for algebraists it is enough to have only is-zero, is-positive, is-negative, suc, pred, neg as basic operations of integers. This standard algebra of integers can be obtained from the recursive term algebra of integers by adding standard operations is-zero, is-positive, is-negative, suc, pred, neg as definable operations and forgetting about basic operations of the recursive term algebra of integers. This process is a special case of the following general construction called hidden function construction (hfc) of an algebra of terms from a recursive term algebra:

--add some structurally recursive operations to a recursive term algebra

--hide some operations of the enriched algebra.

The reason why we allow only structurally recursive operations to be added is because it is not a nice thing to have non-terminating operations as operations of algebras. We also assume that after applying hfc to a recursive term algebra, the resulting algebra of terms still enables us to define the equality check as a structurally recursive operation. This is because algebras without equality check are not interesting. We cannot compute much with it. In case we do not hide any operation in hfc, we call such construction _enrichment._

Now we propose the following definition of data types:

Definition of Data Types

Data types are either recursive term algebras or algebras of terms obtained by hfc.

Note that after applying hfc, we may lose "computability completeness". Thus we have two kinds of data types. A data type which has "computability completeness" is called a _complete data type._ A data type which does not possess this property is called a _non-complete data type._ Obviously all recursive term algebras are complete data types. Curiously the standard algebra of integers is also complete. This completeness is due to the fact that from the standard operations of the standard algebra of integers, by primitive definition, i.e.

-- functional composition

-- definition by cases,

we can define all basic operations of the recursive term algebra of integers. This suggests that both the standard algebra of integers and the recursive term algebra of integers provide equivalent (in fact primitively equivalent) computation basis over the terms of integers. This notion of _primitive equivalence_ of data types has not been studied in the previous theories of abstract data types. We claim that from a computability point of view, this equivalence of data types is important. A technical problem here is that this equivalence is an equivalence between two algebras with different signature.

Even though we may lose "computability completeness" by applying hfc to recursive term algebras, all operations of our data types are still computable, in the sense that we know how to compute them with the aid of "hidden" functions. Therefore all of our data types are computable.

§6. Parameterized Data types

A parameterization of data types can be considered as a "uniform" way of constructing a data type from an argument data type. We treat this concept as recursive definition of terms over the argument data type.

Assume A is an arbitrary data type. We can define a data type S-expression of A as follows:

S-expression of A

(Base of Parameterization)

If a is a term of A then a is an S-expression of A.

(Step of Parameterization)

If S_1 and S_2 are S-expressions of A then so is $(S_1.S_2)$

(Closure of Parameterization)

Nothing is an S-expression of A unless its being so follows from (Base of Parameterization) and (Step of Parameterization).

Then from the closure, we have:

if S is an S-expression of A then either

s is a term of A or

$S = (S_1.S_2)$ for some S-expressions S_1 and S_2.

This gives rise to a basic predicate atom such that:

atom (S) = true if s is a term of A

false otherwise.

Step gives us decomposition operators car, cdr and a composition operator cons such that:

car(s) = s_1 if s = $(s_1.s_2)$ for some s_1 and s_2

undefined otherwise

cdr(s) = s_2 if s = $(s_1.s_2)$ for some s_1 and s_2

undefined otherwise

cons(s_1,s_2) = $(s_1.s_2)$

We also inherit all operations of A. Since equality check in A is a definable operator of A, this operator eq is also a definable operator of S-expression of A.

By structural induction, we can prove the following properties of S-expressions of A.

car(cons(x,y)) = x

cdr(cons(x,y)) = y

cons(car(x),cdr(x)) = x if $\neg$atom(x).

We have observed an example of how a parameterization can be done by recursively defining terms over argument data types. We call this kind of parameterization recursive term parameterization. We allow hfc to be applied to recursive term parameterizations to produce another parameterizations, as we allow hfc to be applied to recursive term algebras to produce a new algebra of terms.

We will explain how this works to specify identifiers over argument data type. Let A be an arbitrary data type. We first define a recursive term parameterization Words of A.

Words of A

(Base of Parameterization)

If a is a term of A then it is a word of A

(Step of Parameterization)

If x is a word of A and a is a term of A, then a.x is a word of A.

(Closure of Parameterization)

Nothing is a word of A unless its being so follows from (Base of Parameterization) and (Step of Parameterization).

Canonically obtained basic operations of words of A allows us to structurally define a predicate is-eq which satisfies:

is-eq(x,y) = true if x and y are the same words
= false otherwise.

Since in the data type of identifiers, we are interested only in the equality check of identifiers, we can apply the following hfc to obtain a parameterization identifiers over A.

Identifiers over A

-- add is-eq to Words of A.

-- forget all basic operations of Words of A.

It is obvious that recursive term parameterizations preserve "computability completeness". For this reason, we call them complete parameterization. Usually application of hfc results in the loss of this property. For example, even though Words of A is complete, Identifiers of A is not complete.

§7. Data Types with Congruence Relation

As pointed out in the end of the first section, there are some structures for each element of which it is impossible or very difficult to provide a unique representation. To cope with this situation, what we have to do is to introduce a congruence relation which identifies different representations of the same object. In initial algebra approach, this relation is specified by equational axioms and this congruence relation is used meta-theoretically to form quotient algebra so that the class of representations denoting the same object forms an equivalence class.

We observe that, the congruence relation must be a total computable (decidable) one to make computational sense. Furthermore, preferably it should be a structurally recursive relation. It is easier to define such relation as a (structurally) definable operation than to define it through equational axioms and make sure they are totally computable (or structurally recursive). In fact the latter is the approach taken by Bergstra & Tucker [13] when they introduced computability to initial algebraic specification method. Furthermore, once we obtain the congruence relation as a definable operation, by enriching the term algebra with this congruence relation as an operation, we get the same effect as taking quotient by the congruence relation, thus we do not need to form quotient algebra. We "internalize" the congruence relation rather than bringing it up to the meta-theoretical level and take quotient as in the initial algebra approach.

Since we know that the data type of finite power set is an example which requires congruence relation, let us examine how term algebra specification method can specify this data type. Basic strategy is to represent a set $\{a_1,a_2,\ldots,a_n\}$ by lists $<a_1,a_2,\ldots,a_n>$, $<a_2,a_3,\ldots,a_n,a_1>$ ..., or $<a_n,a_{n-1},\ldots,a_1>$. Then we identify all of these representations by a congruence relation, which is the permutation relation. Therefore we start with specifying a parameterized data type List(A) by a recursive term parameterization. Using basic operations of this parameterized data type, we define basic set operations in terms of list representations. Then we define an extra operation equalset satisfying:

$$\begin{aligned} \text{equalset } (l_1,l_2) &= \underline{\text{true}} \text{ if } l_1 \text{ is a permutation of } l_2 \\ &= \underline{\text{false}} \text{ otherwise.} \end{aligned}$$

It is easy to check that this relation is a congruence relation to basic set operations. After enriching List(A), with these defined operations, we forget all basic operations of List(A). Then we get a parameterized data type Set(A). In fact this is how LISP [14] treats sets.

This kind of "internalization" of meta-theoretic concept is quite a common exercise in mathematics. For example, in recursive function theory we treat number theoretic functions without embedding it into logic. Thus we can not inherit predicates from logic. To cope with this lack of predicates, which is a meta-theoretical concept to recursive function theory, recursive function theorists "internalize" predicates by considering computable predicates to be computable functions which ranges over $\{0,1\} \subset N$.

§8. Abstractness of Specification

One may criticize our approach for being dependent on term representations, thus lucking in "abstractness" of the specification. The issue of "abstractness" is quite a controversial philosophical issue and we are quite reluctant to take sides. Anyway, for finitarists, "abstractness" is finitely establishable properties of finitely decidable collection of finitely examinable (thus concrete) objects. In this sense, our term algebra specification method is abstract enough already. But from more conventional view of "abstractness", we agree that our approach as it is now lacks in "abstractness". Our answer to this question is the following general definition of abstract data types:

Definition of Abstract Data Types

An _abstract_ _data_ _type_ is a class of algebras which are "constructively" isomorphic to a data type.

Constructive isomorphism plays an essential role here. For example, let A be a finite data type and Ident(A) be the data type of identifiers of A defined as in §6. Then this data type is isomorphic to the algebra PRF of all primitive recursive functions with the extensional equality of functions as its operation. Thus Ident(A) is the same as PRF in non-constructive algebraic sense. But we can not use PRF as a substitute for Ident(A), because in PRF, we can not compute the equality of elements. Unfortunately PRF does not serve as a data type of identifier. In fact PRF and Ident(A) are not constructively isomorphic.

§9. Sub-Data Types

Formally speaking, "recursive definition" of sets is an unambiguous context free grammar. This has the following two important implications:

First, our data types are algebras over context free languages. This restriction is practically reasonable, for even programming languages are context free. Theoretically, an ambiguous context free grammar is important because it provides a computationally complete set of basic operations over the language it generates. If a decidable set is non-context free, there is no obvious way of providing computational basis from a grammar of the set.

Second, algebras generated by unambiguous context free grammars are not just many sorted algebras. Each non-terminal X of a grammar G corresponds to a sort of the algebra Alg(G) generated by G. The carrier set $Alg(G)_X$ of this sort in Alg(G) is the language $L(X) = \{w/X \overset{*}{-} w\}$. Therefore, if G has a production rule $X_1 \to X_2$ then $L(X_1) \subset L(X_1)$. For example, A is a subset of S-expr(A). Our algebras are order sorted algebras of Goguen [14] or classified algebras of Wadge [15] in which a carrier set on to a subset of others. In our data types, polymorphic operations can very easily be introduced. For example, due to $A \subset$S-expr(A), all operations over S-expr(A) are polymorphic.

Emphasizing the importance of context free data types does not mean neglecting the importance of non-context free decidable data structure. In formal language theory, it is well-known that any decidable set X is a decidable subset of some context free set D. Also it is easy to observe that a function f:X->D is computable iff it is the restriction of a computable function f:D->D. This indicates that the computability over X is inherited from that over D, thus the data structure X should be treated within the structure of D. For example, the set Pr of all prime numbers is a non-context free decidable subset of N. All computable operations over Pr are restriction of number theoretic computable operations. Since most number theoretic operations do not preserve primeness, we have to discuss prime numbers inside the theory of natural numbers. If we treat prime numbers independently to natural numbers, as far as the authors know, the only obvious operations of importance over Pr are just the 1st prime number as a constant, the next prime number operation, and the previous prime number operation. Thus the algebra of prime numbers becomes just isomorphic to the algebra of natural numbers.

These observations suggest the following notion of sub-data types:

Definition of Sub-data Types

Let D be a data type and $r=<r_s:D_s->\{\underline{true},\underline{false}\}>$ be a family of definable predicates over carrier sets D_s. A sub-data type of

D determined by r is an algebra of terms obtained by adding $R_s=\{d|r_s(d)=\underline{true}\}$ as subsorts of D_s, r_s as an operation over D_s, and the restriction to R_s of operations over D_s

Algebras obtained from (D,r) by forgetting some operations are also called sub-data types of D (derived from r) as operations over R_s. We write (D,r) to denote this sub-data type. Algebras obtainable from (D,r) by forgetting some operations are also called sub-data types of D (derived from r).

Obviously D is a sub-data type of itself. It is possible that operations over R_s ranges over $R_{s'}$, for some sorts s'. In this case, R_s and operations over them form a closed algebra. Such algebra is called a decidable data type. Algebras obtained from such algebra by forgetting some operations are also called decidable data types. All data types are decidable data types. Order sorted version of computable algebras of Rabin [16] and Malchev [17] are exactly decidable data types. In fact, given any order sorted computable algebra A, we can find a data type D s.t. A is effectively isomorphic to a decidable data type obtained from D and a family r of definable predicates over D.

Sub-data types are not just to introduce non-context free decidable substructures of context free data types. There are many important context free sub-structures of context free data types. For example, in the data type N of natural numbers, we can define a predicate r s.t. $r(x) = \underline{true}$ iff $10 \leq x \leq 20$. Then (N,r) is a PASCAL subrange [10..20]. Even though $R = \{x|r(x) = \underline{true}\}$ is a finite set, (N,r) is not a finite data type as in 4, for the operations associated with the set R in (N,r) and in the finite data type R are different. Finite data types corresponds to PASCAL scalar types. It is an unfortunate confusion in PASCAL that subranges and scalars are treated in the same category, namely the category of "data types". In our understanding, subranges are sub-data types and scalars are data types.

Another advantage of sub-data types is that, using the same idea as in Goguen's order sorted algebras [14], and Wadge's classified algebras [15] we can totalize all operations without introducing errors. Remember that the partiality of operations comes from the incapability of decomposing elements which are in the data type due to the base of the recursive definition. So if the decomposing operations are restricted to non base elements, all operations become error free. But the predicate is-nonbase s.t.

is-nonbase (d) = <u>true</u> if d is not a base element
<u>false</u> if d is a base element

is a primitively definable operation. Therefore for each data type we have an equivalent sub-data type which is error free. For example, in the data type N of natural numbers defined in §1, we can define a predicate.

is-positive (n):= <u>false</u> if is-zero(n) = <u>true</u>
<u>true</u> otherwise.

Then we can define a sub-data type of natural numbers defined by is-positive. This sub-data type (N,is-positive) have the set N of natural numbers as a sort and the set P of positive numbers as a sub-sort of N. It has operations

is-zero: N->{<u>true</u>, <u>false</u>}

suc:N->N

pred:N->N

0:->N

is-positive:N->{true, false}

suc$\restriction$P: P->N

pred$\restriction$P: P->N

is-zero$\restriction$P: P->{true, false}

By forgetting pred, suc$\restriction$P, is-zero$\restriction$P, we get a desired sub-data type. In this sub-data type no operation is partial, thus it is error free.

§10. Concluding Remarks

Lehman-Smyth [18] discussed an extentional analogue. It is known that a continuous functor F:C->C where C is a complete category yields an isomorphism:

$$D\cong F(A).$$

They claimed that this isomorphism yields basic operations to the object D. Wadge [19] pointed out that it is not the isomorphism but the intentional information of F which yields basic operations. We agree with Wadge and think that the reason why our approach worked out is because we added intensional information to F and considered it as an unambiguous context free grammar. In general, given just a continuous functor F:C-C, there is no canonical way to associate basic operations to the fixed point D$\cong$F(A). The other main difference between our approach and Lehman-Smyth is that we discard infinitary data types. This made it possible to develop a clean theory of computability of data types. There was no theory of sub-data types at all in Lehman-Smyth. Also our approach is based on a lot simpler mathematics than theirs.

Mike Levy told us that Burstall et al. [20] used grammar to generate carrier sets of data types in HOPE. They did not treat computability and there is no automatic generation of basic operations in HOPE.

Acknowledgement

We thank Paul Gilmore, Paul Voda, Harvey Abramson and Alan Mackworth of U.B.C. for discussion and valuable suggestions. Thanks are also due to Jose Meseguer, Mike Levy and David Parnas for their interest and discussions.

References

[1] Goguen, Thatcher, Wagner & Wright. Abstract Data Types as Initial Algebras and Correctness of Data Representations. In Computer Graphics, Pattern Recognition and Data Structure, IEEE. 1975.

[2] Goguen, Thatcher, Wagner & Wright. Initial Algebra Approach to Specification, Correctness and Implementation of Abstract Data Types. In R. Yeh (editor), Current Trends in Programming Methodology. Prentice-Hall, 1978.

[3] Goguen & Tardo. An Introduction to OBJ: A Language for Writing and Testing Software Specifications. In Specification of Reliable Software, 1979.

[4] Goguen & Meseguer. An Initiality Primer. to appear

[5] Klaeren, H. An Abstract Software Specification Technique based on Structural Recursion. SIGPLAN Notice 15, No. 3, 1980.

[6] Wirth, N. The Programming Language PASCAL. Acta Informatica 1, 1971.

[7] Peck et al. Revised Report on Algorithm Language ALGOL 68. Springer-Verlag, 1976.

[8] Peck, J. Two Level Grammars in Action. In Information Processing 74 (Stockholm) North Holland, 1974.

[9] Scott, D. Outline of a Mathematical Theory of Computation. Proc. of the 4th Annual Princeton Conference on Information Science and Systems, 1970.

[10] Church, A. The Calculi of Lambda-conversion. Princeton Univ. Press, 1941.

[11] Godel, K. Uber eine bisher noch nicht benntzte Erweiterung des finiten Standpunktes. Dialectica 12, 1958.

[12] Gilmore, P. Combining Unrestricted Abstraction with Universal Quantification, Seldin & Hindley (editors), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980.

[13] Bergstra & Tucker, A Characterization of Computable Data Types by means of a Finite Equational Specification Method. Technical Report, Mathematische Centrum, Amsterdam, Holland, IW124/79, 1979.

[14] Goguen, J. Order Sorted Algebras, Semantics and Theory of computation report No. 14, UCLA, 1978.

[15] Wadge, W. Classified Algebras. Research Report No. 46, Dept. of Computer Science, Univ. of Warwick (1982).

[16] Rabin, M. Computable Algebras: General Theory and Theory of Computable Fields. Transaction of the American Mathematical Society 95, 1960.

[17] Malcev, A.I. Constructive Algebras I. Russian Mathematical Surveys 16(3), 1961.

[18] Lehman-Smyth, Algebraic Specification of Data Types - a synthetic approach, Univ. of Leeds, Dept. of Computer Studies, Report 115, 1978.

[19] W. Wadge, Private Communication, 1978.

[20] Burstall, MacQueen and Sannella, Hope: An Experimental Language, Proc. of POPL, 1981.