POINTERS AS A DATA TYPE

N.C.K. Phillips and S.W. Postma
University of Natal
Pietermaritzburg 3200

1. INTRODUCTION

The distinction between an abstract data type and an implementation of a data type is not as clear as is commonly thought. A hash table can be used to implement the type bag but few of our colleagues think of a hash table as itself being a data type. Similarly, pointers can be used to give a linked implementation of the data type list, but few of us seem to think of pointers as themselves forming a data type. However we shall give an algebraic specification of a pointer data type which is an abstraction of the representation of pointers in Pascal. The usual linked structures can then be specified abstractly via mutually recursive type specifications. In addition we shall describe a uniform method of constructing implementations of data type specifications via operations on strings.

2. THE TYPE POINTER

We give below an algebraic specification of the type pointer. Note that this type has two parameters, Heap and Item. The specification is sufficiently complete in the sense of Guttag [1] and safe in the sense of Phillips [8]. This implies that equality between pointers is implicitly considered to be a boolean valued operator of the type and that the axioms determine (up to isomorphism) a unique model whose objects of type pointer can be named by terms of the word algebra of the specification.

Type (Heap of) Pointer (to Item):

nil	→ Pointer
new(heap)	→ Pointer
mkpointto(pointer,item)	→ Pointer U {undefined}
pointsto(pointer)	→ Item U {undefined}
is-nil(pointer)	→ Boolean
is-new(pointer)	→ Boolean

For p,q in Pointer; g,h in Heap and i,j,k,l in Item:

{action of is-nil}
1.1) is-nil(nil) = true
1.2) is-nil(new(h)) = false
1.3) is-nil(mkpointto(p,i)) = if is-nil(p) then undefined else false

{action of is-new}
2.1) is-new(nil) = false
2.2) is-new(new(h)) = true
2.3) is-new(mkpointto(p,i)) = if is-nil(p) then undefined else false

{action of pointsto}
3.1) pointsto(nil) = undefined
3.2) pointsto(new(h)) = undefined
3.3) pointsto(mkpointto(p,i)) = if is-nil(p) then undefined else i

{action of equality between pointers}
4.1) mkpointto(nil,i) = undefined
4.2) mkpointto(mkpointto(new(h),i),j) = mkpointto(new(h),j)
4.3) p=nil = is-nil(p)
4.4) new(h) = new(g) = h =g
4.5) mkpointto(p,i) = mkpointto(q,j) = if is-nil(p) or is-nil(q) then undefined else p = q and i = j

quæstiones informaticæ, 5, 3, pp 21-23, issn 0254-2757

Some brief explanation will help comprehension of the axioms. New obtains new pointers in one-to-one correspondence with the elements of Heap. mkpointto(p,i) corresponds to Pascal's p^:=i. Pointsto(p) corresponds to Pascal's p^. Given sets Heap and Item, the data type can be modelled in an intuitively natural way, with the set of pointers taken to be {nil} ∪ {(h;undefined): h ∈ Heap} ∪ {(h,i) : h ∈ Heap,i ∈ Item}. In this model, pairs (h,undefined) play the role of new pointers and a pair (h,i) points to i. The operator new is implemented in the model by new(h) = (h,undefined). Some further discussion of the axioms is given in section 3 below.

The extent of the role that formal specifications should play in developing computer applications has been argued widely. An excellent discussion of the issue is contained in [2]. However nobody denies that having to meet a formal specification reduces the possibility of implemention dependent results. This can be illustrated with our type Pointer. Neither UCSD Pascal IV.1 nor Turbo Pascal 3.0 conform to axiom 7 of our specification. The program below was run under both systems.

```
program notsotrite(output);
var p:^integer;
begin
p:= nil;
writeln(p^);
end.
```

The UCSD program wrote 0, and Turbo program wrote 4146. However in HP 3000 Pascal the program aborted with the message 'error #24 : bounds violation'.

3. MODELS AND IMPLEMENTATIONS

The construction of correct specifications which adequately reflect and refine our intuition about a data type has turned out to be a tricky task for the human mind. Reasons for this and a design methodology to alleviate it are given in [3]. The difficulties have led to the production of automated systems for the interactive design and testing of specifications. OBJ [4] is such a system for data type specifications and AFFIRM [5] has also been used for this purpose. One can also attempt automatic generation of data type implementations from specifications and this has been described in [6] and [7]. Of course such automatically generated implementations are not likely to be efficient.

We mentioned earlier that our axioms determine (up to isomorphism) a unique model whose objects of the type specified are named by terms of the word algebra of the specification. This is in fact a general property of consistent specifications which are safe (but specifications which are sufficiently complete but not safe need not have this property). Indeed, any model which satisfies our specification contains this unique model and again this is true generally for safe specifications. Let us call this model the canonical model. Corresponding to it there is a general method of implementation which we shall now describe for our example Pointer.

The objects which represent pointers are those strings in the word algebra which are terms of type Pointer. More precisely, the set of these objects is the set T defined below.

(i) nil is in T
(ii) if h is in Heap then new(h) is in T,
(iii) if t is in T and i is in Item then mkpointto(t,i) is in T,
(iv) only the strings given by (i) - (iii) above are in T.

Now axioms 1.1 - 1.3 of the specification can be taken to be recursive definitions of is-nil on T. Axioms 2.1 - 2.3 recursively define is-new on T and axioms 3.1 - 3.3 define pointsto. Axioms 4.1 - 4.5 determine a binary relation on T which is to be taken as the implementation of equality between pointers. Only a modicum of ingenuity is needed to program an equality test which conforms to axioms 4.1 - 4.5.

Let us call this implementation the canonical implementation. Algebraists have terminology which aptly describes the relationship between canonical implementations and canonical models. The equality relation in the canonical implementation of a safe specification is a congruence relation with respect to the other operators, hence we can lift these operators to the congruence classes. The resulting algebra is the canonical model for the specification.

4. LINKED STRUCTURES

In their seminal paper [1], Guttag and Horning state: "Theoretically, it would be possible to define several types at once via mutual recursion. It seems, however, that in general a clean separation of types leads to clearer specifications," We are not aware of any previously published examples of mutually recursively specified data types. Yet it is precisely this mutual recursion which makes the type pointer useful. Below we give a specification of a type Node. Node's parameters are Item , and Pointer with parameters Heap and Node itself.

Type Node (of Item, (Heap of) Pointer (to Node)):

data(node)	→ Item
next(node)	→ (Heap of) Pointer (to Node)
chgdata(node,item)	→ Node
chgnext(node,pointer)	→ Node

For n,m in Node, p in Pointer and i in Item:

1) data(chgdata(n,i)) = i
2) next(chgdata(n,i)) = next(n)
3) data(chgnext(n,p)) = data(n)
4) next(chgnext(n,p)) = p
5) n=m = data(n) = data(m) and next(n) = next(m)

The axioms simply formalize our intuition that chgdata(n,i) and chgnext(n,p) behave like Pascal's n^.data:= i and n^.next := p and that a node is determined by its data and next fields. The specification formalizes nodes in a linked list of Item and such a linked list is formally a Pointer to a Node. Other familiar linked structures such as linked trees can be obtained by altering the specification of Node.

While the notion of 'list' seems more abstract than that of 'linked list', we have shown above that linked lists can be specified as abstractly as lists. Thus in the area of specification there are shades of meaning to the term 'abstract' and perhaps abstraction,like beauty, lies largely in the eye of the beholder.

5. REFERENCES

1 Guttag J.V. & Horning J.J., [1978], The algebraic specification of abstract data types, *Acta Informatica*, **10**, 27-52.

2 Guttag J., Horning J & Wing J., [1982], Some notes on putting formal specifications to productive use, *Science of Computer programming*, **2**, 53-68.

3 Guttag J.V., Horowitz E. & Musser D.R., [1978], The design of data type specifications, in *Current Trends in Programming methodology*, Vol IV: Data Structuring, 60-79, Prentice-Hall, Englewood Cliffs, N.J..

4 Goguen J.A. & Tardo J.J., [1979], An introduction to OBJ: A language for writing and testing formal algebraic specifications, in *Proc. Specifications of Reliable Software*, 170-189.

5 Musser D.R., [1979], Abstract data type specification in the Affirm system, in *Proc. Specifications of Reliable Software* , 47-57.

6 Guttag J.V., Horowitz E. & Musser D.R., [1978], Abstract data types and software validation , *Communications of the ACM*, **21**, 1048-1064.

7 Moitra A., [1982], Direct implementation of algebraic specification of abstract data types. *IEEE Transactions on software Engineering*, **8**, 1, 12-20.

8 Phillips, N.C.K, [1984], Safe data type specifications, *IEEE Transactions on Software Engineering*, **10**, 3, 285-289.