This is a preprint of a paper to be given at the Conference on New Directions in Algorithmic Languages sponsored by IFIP Working Group 2.1, Munich, August 1975.

USER-DEFINED TYPES AND PROCEDURAL DATA STRUCTURES AS COMPLEMENTARY APPROACHES TO DATA ABSTRACTION

John C. Reynolds

Syracuse University, Syracuse, New York

ABSTRACT User-defined types (or modes) and procedural (or functional) data structures are complementary methods for data abstraction, each providing a capability lacked by the other. With user-defined types, all information about the representation of a particular kind of data is centralized in a type definition and hidden from the rest of the program. With procedural data structures, each part of the program which creates data can specify its own representation, independently of any representations used elsewhere for the same kind of data. However, this decentralization of the description of data is acheived at the cost of prohibiting primitive operations from accessing the representations of more than one data item. The contrast between these approaches is illustrated by a simple example.

Work supported by National Science Foundation Grant GJ-41540

Introduction

User-defined types and procedural data structures have both been proposed as methods for data abstraction, i.e., for limiting and segregating the portion of a program which depends upon the representation used for some kind of data. In this paper we suggest, by means of a simple example, that these methods are complementary, each providing a capability lacked by the other.

The idea of user-defined types has been developed by Morris,[1,2] Liskov and Zilles,[3] Fischer and Fischer,[4] and Wulf[5], and has its roots in earlier work by Hoare and Dahl.[6] In this approach, each particular conceptual kind of data is called a type, and for each type used in a program, the program is divided into two parts: a type definition and an "outer" or "abstract" program. The type definition specifies the representation to be used for the data type and a set of primitive operations (and perhaps constants), each defined in terms of the representation. The choice of representation is hidden from the outer program by requiring all manipulations of the data type in the outer program to be expressed in terms of the primitive operations. The heart of the matter is that any consistent change in the data representation can be effected by altering the type definition without changing the outer program.

Various notions of procedural (or functional) data structures have been developed by Reynolds,[7] Landin,[8] and Balzer.[9] In this approach, the abstract form of data is characterized by the primitive operations which can be performed upon it, and an item of data is simply a procedure or collection of procedures for performing these operations. The essense of the idea is seen most clearly in its implementation: an item of procedural data is a kind of record called a closure which contains both an internal representation of the data and a pointer (or flag field) to code for procedures for manipulating this representation. A program with access to a closure record is only permitted to examine or access the internal representation by executing the code indicated by the pointer, so that this code serves to close off or protect the internal representation.

In comparison with user-defined types, procedural data structures provide a decentralized form of data abstraction. Each part of the program which creates procedural data will specify its own form of representation, independently of the representations used elsewhere for the same kind of data, and will provide versions of the primitive operations (the components of the procedural data item) suitable

for this representation. There need be no part of the program, corresponding to a type definition, in which all forms of representation for the same kind of data are known. But a price must be paid for this decentralization: a primitive operation can have access to the representation of only a single data item, the item of which the operation is a component.

Apparently this price is inevitable. If an operation is to have access to the representation of more than one item of data, each of which may have several possible representations, then its definition cannot be "decentralized" into one part for each representation, since one must provide for every possible combination of representations. Presumably this requires the definition to occur at a point in the program where all possible representations of the operands are known.

Linguistic Preliminaries

Before illustrating these ideas, we must digress to explain (informally) the language we will use. It is an applicative language, similar to pure LISP[10] or the applicative subsets of GEDANKEN,[7] PAL,[11] or ISWIM,[12] but with a complete type structure somewhat like Algol 68.[13] Types will be indicated by writing $\varepsilon\ \tau$, where τ is a type expression, after binding occurrences of identifiers (except where the type is obvious from context). Type expressions are constructed with the operators $\rightarrow$ denoting functional procedures, $\times$ denoting a Cartesian product, and + denoting a named disjoint union.

The named disjoint union is sufficiently novel to require a more detailed explanation. If $\tau_1, \ldots, \tau_n$ are type expressions denoting the sets $S_1, \ldots, S_n$ and $i_1, \ldots, i_n$ are distinct identifiers, then

$$i_1\colon \tau_1 + \ \ldots\ + i_n\colon \tau_n$$

is a type expression denoting the set of pairs

$$\{\langle i_k, x\rangle \mid 1 \leq k \leq n \text{ and } x \in S_k\}\ .$$

If e is an expression of type τ_k with value x, then

$$\underline{\text{inject}}\ i_k\ e$$

is an expression of type $i_1\colon \tau_1 + \ldots + i_n\colon \tau_n$ with value $\langle i_k, x\rangle$.

Let e be an expression of type $i_1: \tau_1 + \ldots + i_n: \tau_n$ with value <i, x>, let $i_{k_1}, \ldots, i_{k_m}$ be distinct members of the set of identifiers $\{i_1, \ldots, i_n\}$, for $1 \leq j \leq m$ let ℓ_j be an expression of type $\tau_{k_j} \rightarrow \tau'$ with value f_j, and let e' be an expression of type τ' with value x'. Then

$$\underline{\text{unioncase}}\ e\ \underline{\text{of}}\ (i_{k_1}: \ell_1, \ \ldots\ , i_{k_m}: \ell_m, \underline{\text{other}}: e')$$

is an expression of type τ' with the value

$$\underline{\text{if}} \begin{bmatrix} i = i_{k_1} \\ \vdots \\ i = i_{k_m} \\ \text{otherwise} \end{bmatrix} \underline{\text{then}} \begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \\ x' \end{bmatrix}$$

When m = n, the other clause will be omitted.

We use the type expression nilset to denote a standard one-element set, whose unique member is denoted by ().

Integer Sets as a User-Defined Type

Our example is an implementation of the abstract concept of sets of integers. Using the approach of user-defined types, we wish to define a type *set* and primitive constants and functions

none ε set

all ε set

limit ε integer × integer × set → set

union ε set × set → set

exists ε integer × integer × set → Boolean

satisfying the specifications

```
none = {}
all = The set of all (machine-representable) integers
limit(m, n, s) = s ∩ {k | m ≤ k ≤ n}
union(s1, s2) = s1 ∪ s2
when m ≤ n, exists(m, n, s) = (∃ k) m ≤ k ≤ n and k ε s
```

To make our solution seem more realistic, we require that the execution of *limit* and *union* should require time and space bounded by constants which are independent of their arguments. Of course this will exact a price in the speed of *exists*.

An appropriate and simple solution is to represent a set by a list structure which records the way in which the set is constructed via primitive operations. Thus the representation of a set is a disjoint union, over the four set-valued primitive functions (including constants), of sets of possible arguments for these functions. More precisely, this representation is defined by the recursive type declaration:

```
set = nonef: nilset + allf: nilset + limitf: integer × integer × set
        + unionf: set × set
```

and the effect of *none*, *all*, *limit*, or *union* is to imbed its arguments into the appropriate kind of list element:

```
none = inject nonef ()
all = inject allf ()
limit(m, n, s) = inject limitf (m, n, s)
union(s1, s2) = inject unionf (s1, s2)
```

(Roughly speaking, we are representing sets by a free algebra with constants *none* and *all*, and operators *limit* and *union*.) The entire computational burden of interpreting this representation falls upon the function *exists*:

```
exists(m, n, s) = unioncase s of
    (nonef: λ(). false,
    allf: λ(). true,
    limitf: λ(m1, n1, s1). max(m,m1) ≤ min(n,n1)
              and exists(max(m,m1), min(n,n1), s),
    unionf: λ(s1, s2). exists(m, n, s1) or exists(m, n, s2) )
```

(We assume that the operations and and or do not evaluate their second operand when the first operand is sufficient to determine their result.)

Although the above is a definition of the type *set* which meets our specifications, it can be easily improved, even within the time and space constraints imposed upon *limit* and *union*. For example, both *limit* and *union* can be optimized by taking advantage of some obvious properties of sets - the result of *limit* can be simplified when its last argument is *none* or another application of *limit*, and the result of *union* can be simplified when either argument is *none* or *all*:

```
limit(m, n, s) = unioncase s of
     (nonef: λ(). none,
     limitf: λ(m1, n1, s1). if max(m, m1) ≤ min(n, n1)
               then inject limitf (max(m,m1), min(n,n1), s1) else none,
     other: inject limitf (m, n, s)  )
union(s1, s2) = unioncase s1 of
     (nonef: λ(). s2, allf: λ(). all,
     other: unioncase s2 of
          (nonef: λ(). s1, allf: λ(). all,
          other: inject unionf (s1, s2) ))
```

In conclusion, we show how our specification of integer sets might be "packaged" in a language permitting user-defined types:

```
newtype set = nonef: nilset + allf: nilset + limitf: integer × integer × set
     + unionf: set × set
with none ε set = inject nonef (),
     all ε set = inject allf (),
     limit ε integer × integer × set → set =
          λ(m, n, s).unioncase s of
          (nonef: λ(). none,
          limitf: λ(m1, n1, s1). if max (m,m1) ≤ min(n,n1)
               then inject limitf (max(m,m1), min(n,n1), s1) else none,
          other: inject limitf (m, n, s) ),
     union ε set × set → set =
          λ(s1, s2). unioncase s1 of
          (nonef: λ(). s2, allf: λ(). all,
          other: unioncase s2 of
               (nonef: λ(). s1, allf: λ(). all,
               other: inject unionf (s1, s2) )),
     exists ε integer × integer × set → Boolean =
          λ(m, n, s). unioncase s of
          (nonef: λ(). false,
          allf: λ(). true,
          limitf: λ(m1, n1, s1). max(m,m1) ≤ min(n,n1)
                    and exists(max(m,m1), min(n,n1), s),
          unionf: λ(s1, s2). exists(m, n, s1) or exists(m, n, s2) )
in <outer program>
```

The language used here is an outgrowth of the ideas discussed in reference 14. A complete exposition of this language is beyond the scope of this paper, but the following salient points should be noted:

(1) The type declaration between newtype and with binds all occurrences of the type identifier *set* throughout the above expression (including occurrences in <outer program>). The ordinary declarations between with and in bind all occurrences of the ordinary identifiers *none*, *all*, *limit*, *union*, and *exists* throughout the expression.

(2) With regard to occurrences of *set* between with and in, the type declaration behaves like a mode definition in Algol 68, i.e., *set* is equivalent to the type expression on the right side of the type declaration, and the type-correctness of the text in with ... in depends upon this type expression.

(3) In <outer program> occurrences of *set* behave like a primitive type, e.g., integer or Boolean. In other words, <outer program> must be a correctly typed expression regardless of what type expression might be equivalent to *set*. This insures that all manipulations of the user-defined type in <outer program> must be expressed in terms of the primitives declared in with ... in.

(4) Although it is not illustrated by our example, it should be possible to declare simultaneously several related user-defined types between newtype and with. This ability is needed to permit the definition of multiargument primitive functions which act upon more than one user-defined type. An example might be the use of the types *point* and *line* in a program for performing geometrical calculations.

Integer Sets as Procedural Data Structures

We now develop integer sets as procedural data structures. The starting point is the realization that all we ever want to do to a set s, aside from using it to construct other sets, is to evaluate the Boolean expression $exists(m, n, s)$. This suggests that we can simply equate the set s with the Boolean function $\lambda(m, n).\ exists(m, n, s)$ which characterizes the only information we want to extract from the set.

Thus we define

$$\text{set} = \underline{\text{integer}} \times \underline{\text{integer}} \rightarrow \underline{\text{Boolean}}$$

and specify that if $s \in set$ represents the "mathematical" set s_0, then for $m \leq n$,

$$s(m, n) = (\exists k)\ m \leq k \leq n \text{ and } k \in s_0.$$

The need for defining the primitive function *exists* has vanished since this function has been internalized - its value for a particular *set* is simply the (only component of the) *set* itself. The remaining primitive constants and functions are easily defined by:

```
none = λ(m, n). false
all = λ(m, n). true
limit(m, n, s) = λ(m1, n1).
     max(m,m1) ≤ min(n,n1) and s(max(m,m1), min(n,n1))
union(s1, s2) = λ(m, n). s1(m, n) or s2(m, n)
```

In this approach, there is no "outer program" from which the definition *set* = integer × integer → Boolean is hidden. Any part of the program can create a *set* by giving an appropriate function whose internal representation (the collection of values of global variables which form the fields of the closure record) can be arbitrary. For example, in augmenting an existing program, one might write

$$\lambda(m, n).\ \text{even}(m)\ \underline{\text{or}}\ (m < n)$$

to denote the set of even integers, or

$$\underline{\text{letrec}}\ s = \lambda(m, n).\ (m \leq n)\ \underline{\text{and}}\ (p(m)\ \underline{\text{or}}\ s(m+1, n))\ \underline{\text{in}}\ s$$

to denote the set of integers satisfying the predicate p. The procedural approach insures that these definitions will mesh correctly with the rest of the program, even though they introduce novel representations.

This kind of extensional capability, which is the main advantage of the procedural approach, is offset by two limitations. In the first place, although (ignoring computability considerations) every set can be represented by a function in integer × integer → Boolean, the converse is false. To represent a set, a function s must satisfy

$$s(m, n) = \bigvee_{k=m}^{n} s(k, k)$$

for all m and n such that $m \leq n$. This kind of condition, which cannot be checked syntactically, must be satisfied by all parts of the program which create sets.

A more important limitation is that only the function *exists*, which has been internalized as (the only component of) a procedural data item, is truly primitive in the sense of having access to the internal representation of a set. Essentially, we have been forced to express the functions *limit* and *union* in terms of the internalized *exists*. We are fortunate that our example permits us to do this at all. Even so, we are prevented from optimizing *limit* and *union* as we did in the user-defined-type development. There is no practically effective

way that *limit*(*m*, *n*, *s*) can "see" whether *s* has the form *none* or *limit*(*m1*, *n1*, *s1*), or that *union*(*s1*, *s2*) can "see" whether *s1* or *s2* has the form *none* or *all*.

In fact, this difficulty can be surmounted for *limit* but not for *union*. The solution is to internalize *limit* as well as *exists*, so that both functions have access to internal representations. Thus we represent sets by pairs of functions:

$$\text{set} = (\underline{\text{integer}} \times \underline{\text{integer}} \to \underline{\text{Boolean}}) \times (\underline{\text{integer}} \times \underline{\text{integer}} \to \text{set})$$

and specify that if *s* represents the mathematical set s_o then for $m \leq n$,

$$s.1(m, n) = (\exists k)\ m \leq k \leq n \text{ and } k \in s_o\ ,$$

and for all *m* and *n*, *s.2*(*m*, *n*) represents the mathematical set

$$s_o \cap \{k \mid m \leq k \leq n\}$$

(Here *s.1* and *s.2* denote the components of the pair *s*.)

In this approach, we may define *none* by:

```
none = (λ(m, n). false, λ(m, n). none)
```

Note the peculiar kind of recursion which is characteristic of this style of programming: the second component of *none* is a function which does not call itself but rather returns itself as a component of its result.

To define *all* and *union* we first define an "external" *limit* ε integer × integer × *set* → *set* which will be called upon by the internal limiting functions (i.e., the second components) of *all* and *union*:

```
limit(m, n, s) =
     (λ(m1, n1). max(m,m1) ≤ min(n,n1) and s.1(max(m,m1), min(n,n1)),
     λ(m1, n1). if max(m,m1) ≤ min(n,n1) then
                    limit(max(m,m1), min(n,n1), s) else none)
```

Then

```
all = (λ(m, n). true, λ(m, n). limit(m, n, all))
union(s1, s2) = (λ(m, n). s1.1(m, n) or s2.1(m, n),
     λ(m, n). limit(m, n, union(s1, s2)) )  .
```

With these definitions, the internal limiting functions perform simplifications analogous to those performed by *limit* in the user-defined-type approach. Indeed, if one examines the behavior of the closures which would represent sets in an implementation of this definition, one finds that they mimic the list structures of the type

approach almost exactly (except for the simplifications performed by union).

But even to someone who is experienced with procedural data structures, the internalization of *limit* is more a tour de force than a specimen of clear programming. Moreover, internalization cannot be applied to give a function such as *union* access to the internal representation of more than one argument, i.e., we could convert *union*(*s1*, *s2*) to a component of *s1* or of *s2* but not both.

Conclusions

In comparison with user-defined types, procedural data structures offer a more decentralized method of data abstraction which precludes any interaction between different representations of the same kind of data. This offers the advantage of easier extensibility at the price of prohibiting primitive operations from accessing the representations of more than one data item.

Of course, the two approaches can be combined. For example, we can augment our user-defined-type definition to include an additional primitive *functset* ε (integer × integer → Boolean) → *set* which accepts a functional set (in the sense of the first part of the previous section) and produces an equivalent value of type *set*. It is sufficient to add one more kind of record to the disjoint union defining *set* and one more alternative to the branches defining *exists*:

```
newtype set = ... + functsetf: (integer × integer → Boolean)
with ...
     functset ε (integer × integer → Boolean) → set = λf. inject functsetf f,
     exists ε integer × integer × set → Boolean =
          λ(m, n, s). unioncase s of
          ( ... functsetf: λf. f(m, n) )
in <outer program>
```

However, this kind of combination is hardly a unification. To some extent, the data-representation structuring approach of Hoare and Dahl[6] unifies the concepts of user-defined types and procedural data structures, but only at the expense of combining their limitations. It appears that this is inevitable; that the two concepts are inherently distinct and complementary.

The reader should be cautioned that this is a working paper describing ongoing research. In particular, the linguistic constructs we have used are tentative and will require considerable study and evolution before they can be integrated into a complete programming language. The extension of these constructs to languages with imperative features is a particularly murky area.

REFERENCES

1. Morris, J. H., Types are not Sets. Proc. ACM Symposium on Principle of Programming Languages, Boston 1973, pp. 120-124.
2. Morris, J. H., Towards More Flexible Type Systems. Proc. Colloque sur la Programmation, Lecture Notes in Computer Science 19, Springer-Verlag 1974, pp. 377-384.
3. Liskov, B. H. and S. Zilles, "Programming with Abstract Data Types," ACM SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 50-60. (Also available as MIT Project MAC Computation Structure Group Memo 99).
4. Fischer, A. E., and Fischer, M. J., Mode Modules as Representations of Domains. Proc. ACM Symposium on Principles of Programming Languages, Boston 1973, pp. 139-143.
5. Wulf, W., Alphard: Toward a Language to Support Structured Programs, Department of Computer Science Internal Report, Carnegie-Mellon University, Pittsburgh, Pa., April 1974.
6. Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press 1972.
7. Reynolds, J. C., GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. Comm ACM 13 (May 1970), 308-319.
8. Landin, P. J., A Correspondence Between ALGOL 60 and Church's Lambda-Notation. Comm ACM 8 (February-March 1965), 89-101 and 158-165.
9. Balzer, R. M., Dataless programming. Proc. AFIPS 1967 Fall Joint Comput. Conf. Vol. 31, MDI Publications, Wayne, Pa., pp. 535-544.
10. McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Pt. I. Comm ACM 3, 4 (Apr. 1960), 184-195.
11. Evans, A., PAL - A language designed for teaching programming linguistics. Proc. ACM 23rd Nat. Conf. 1968, Brandin Systems Press, Princeton, N.J., pp. 395-403.
12. Landin, P. J., The next 700 programming languages. Comm ACM 9, 3 (Mar. 1966), 157-166.
13. van Wijngaarden, A. (Ed.), Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. Report on the algorithmic language ALGOL 68. MR 101, Mathematisch Centrum, Amsterdam, Feb. 1969.
14. Reynolds, J. C., Towards a Theory of Type Structure. Proc. Colloque sur la Programmation, Lecture Notes in Computer Science 19, Springer-Verlag 1974, pp. 408-423.