Abstract Data Types in the MODEL Programming Language

Robert T. Johnson and James B. Morris

This article is work performed under the auspicies of the United States Energy Research and Development Administration

Edited by Ray D. Tanner

Abstract

The concept of an abstract data type is available in the Model programming language as a proposed improvement to current ideas of programming methodology. In structured programming the principal technique is refinement of procedures. In Model, the analogue is refinement of data types. An abstract data type consists of a data structure and an associated set of operations. The characteristics and suggested uses for this mechanism are discussed. Also presented are several examples culminating in a parallel version of the Fast Fourier Transform.

Keywords and phrases

Data types, abstract data types, programming abstraction, level of abstraction, structured programming, parameterized types, reliable programming.

I. Introduction

The Model programming language is currently in use at the Los Alamos Scientific Laboratory. Model is primarily a programming tool used to investigate the fundamental concepts and requirements of abstract data types. An abstract data type is a specific extension of the basic data types of a programming language. The new data types provided by this extension are usually more specific to a particular programming application.

The concept of abstraction is well known to mathematicians. In general, it consists of the replacement of many interrelated concepts with a single concept. References may then be made to the single (abstract) concept which, in turn, refers to the interrelated concepts. This works as a form of shorthand, or self-encompassing notation.

Abstraction in the control structure of programming languages has existed since the first high-level language: FORTRAN. This abstraction is nothing more than the concept of a "subroutine" or "procedure." It has proved highly successful in classical programming as well as forming a cornerstone of modern programming theory [2].

Data type abstraction is a more contemporary idea and corresponds closely to control structure abstraction. Data type abstraction provides a mechanism for allowing abstraction of data structures in a manner analogous to the formation of subroutines or procedures. The earliest example of data type abstraction in a programming language is the class in the SIMULA programming language. More contemporary thinking [8], however, is that the SIMULA class does not completely satisfy the requirements for a data type abstraction mechanism, primarily because it is not restrictive enough, rather than being too restrictive. Additional theory related to data type abstraction can be found in work by others. Hoare [4,5] has introduced many of the fundamental ideas. Liskov has contributed a survey [8] as well as an independent research project, the CLU programming language [9], in which she introduces "clusters" as data-type extension mechanisms. Geschke and Mitchell [3] and Wulf [18] describe other approaches. J.H. Morris [11,12] and Parnas [13] have contributed to the theory in several papers. Wells [16] has implemented a form of abstraction in Madcap 6.

Following are four basic motivating factors for the use of abstract data types in programming and programming languages.

1. Refinement of Data Type Information

Data type information refinement is one of the most useful characteristics of abstract data types in Model. These abstract data types are termed spaces. To illustrate how refinement can enable the compiler to produce more complete error diagnostics, consider the following example:

A program is written containing 3 tables: T1, T2, and T3. The entries in each of the tables may or may not have different data structures. Assuming they are different, we will designate these data structures S1, S2 and S3. Each of the structures contain a field that is a real number having a different interpretation in each of the three entries. Suppose that we

declare (using a notation in Model similar to one in Pascal[17])

```
type p is real;
type q is real;
type r is real;
```

and we declare entries

```
type S1 is record {... f1:p ...}
type S2 is record {... f2:q ...}
type S3 is record {... f3:r ...}
```

and tables

```
array { S1, [1...100] } T1;
array { S2, [1...100] } T2;
array { S3, [1...100] } T3;
```

The compiler can give only a limited amount of diagnostics in type-checking with regard to usage of the tables. For example, if z has been declared to be of data type r then an assignment statement of the form

```
T2[i].f2 := z
```

cannot be detected by the compiler as an error. As far as the compiler is concerned p, q, and r data types, in actuality, are real data types.

However, declarations

```
space p def real;
space q def real;
space r def real;
```

instead of those for p, q, and r above will result in three new data types which are truly different from real. With all other declarations being the same, the assignment statement above will be detected to be in error by the compiler since an attempt is being made to store an "r" into a "q". Thus, real has been refined into data types p, q, and r.

2. Protection of Structures

SIMULA [1] has a data abstraction facility in a form which still appears very advanced. The class mechanism permits definition of records with associated procedures. The mechanism, however, does not provide for the isolation of the record fields necessary to guarantee the integrity of the structure. References can be made directly to the fields in the record, bypassing the access procedures defined in the class. The SIMULA compiler has no way of assuring that programs maintain the properties that are assumed by the procedures in the class. SIMULA must depend on run-time checks for error detection.

In Model, access to the representation of a new data type is restricted to the operations and procedures declared in the space. This isolates the definition of the data type to the space declaration and enables the compiler to guarantee that the operations defined for the space are not subverted.

3. Ease of Structure Modification

When designing a program it is often not clear which abstract data type will turn out to be the best. It may also be true that during the evolution of the program different representations may be appropriate. For example, the principal goals early in the program's existence may be clarity and ease of implementation. Later, heavy usage may require that efficiency become more important. This change in goals can cause different implementation strategies. A change in the representation of a data abstraction should not cause a change in the programs using the abstraction.

Using the space mechanism in Model, a programmer is assured that the use of one implementation of a data abstraction will not prevent the use of a different implementation strategy later.

The space mechanism encourages the programmer to try to identify abstract data types appropriate for the application. In contrast, most popular programming languages discourage explicit abstraction identification by sanctions such as notational encumberances and efficiency penalties. Model sacrifices neither notational convenience nor implementation efficiency in providing the data type extension mechanism.

4. Program Factorization for Proofs of Correctness

An increasingly popular view of programming methodology conceives the programming process as a sequence of stages. In each stage the program representation consists of a combination of a few abstract operations on a few abstract data types. The successors to a stage will define the operations of an abstract data type in terms of operations for less abstract, more representation-specific data types. Eventually, this process will terminate when all representation decisions have been made and the operations are those in the base programming language.

This is a narrower view than is usually associated with "top-down" or structured programming methodology. There the deferred "operations" are usually more like procedures. Their interface with the calling program is not usually limited to some set of objects which together can be

identified as an abstract entity. Thus, the extent of side effects and independence of calling context of these operations is not nearly so severely controlled.

As Parnas [13] and others have illustrated, the difficulty of verifying that an elaboration of an operation is correct is dependent on the width of the interface between the elaboration and its invocations. This width is measured not only by the amount of information that must be passed through the interface, but by the complexity of the interface description. For example, in a call to a procedure which contains no global references, the interface description is complete in descriptions of the formal parameters and the effects of the procedure on them. In block structured languages, the effects of procedure calls on program variables (side effects) are not limited to the actual parameters. This is because of scope rules that govern the accessibility of global variables; in FORTRAN, common blocks have similar accessibility.

Because of the nature of programming with data abstractions, a language designer cannot simply prohibit the use of side-effect-causing constructs in procedure and function calls and claim to have solved the problem. The problem of the complexity of side effects *is* the problem that the programmer is trying to simplify by constructing the abstract data type. It is quite often true that in an implementation of a data abstraction, an operation that seems only value-producing (in terms of the abstraction), in fact has many global effects at lower representational levels. A good example is the effect on storage management of an operation that results in the allocation of a new block of storage.

To a great extent, the Model programming process consists of elaborating functional (or value-producing) operations for an abstract data type. The elaboration is done by designing side-effect producing operations in such a way that the properties that define the data abstraction are maintained by the operations. This, together with the isolation of the data representation that is guaranteed by the Model compiler, considerably simplifies the program verification process.

In the following sections we illustrate abstract type definitions in Model with some annotated examples. The paper is concluded by presenting basic requirements, problems, and some future directions.

II. Example 1 -- sets of integers over a range

This example has appeared in several previous publications [4,8] to demonstrate some particular aspect of abstract data types. The Model version of sets of integers is demonstrated here partly for comparison with related work and partly because it forms an integral part of the culminating example in this paper.

Our abstract data type for implementation of sets of integers over some range has the classical mathematical properties of sets:

(a) any integer appearing in a set is unique in that set, and
(b) random access to (indexing over) a set is disallowed.

Following is the definition of the abstract data type in the Model programming language. The NOTES following the definition will facilitate reader understanding of the definition mechanisms. While going through the example, keep in mind that Model is similar to Pascal except for syntactic differences.

Boxed numbers appearing in the right margin of the programs below will aid the reader in referring between NOTES and programs.

```
space intset(r) def concrset;                                    1,3
  type concrset is record(size:sizerng;                          2
                          iset:array(r,arraybnd));
  type sizerng is [0...ubnd r - lbnd r + 1];                     4
  type arraybnd is [1...ubnd r - lbnd r + 1];
  type srchbnd is [1...ubnd r - lbnd r + 2];

  insert is <<                                                   5
    formal intset(r) a (varies); r i noresult;
    $  add element i to set a
    if not i in a then                                           6
      #a.size := #a.size + 1;
      #a.iset[#a.size] := i;
    fi;
  >>;

  remove is <<
    formal intset(r) a (varies); r i noresult;
    $  remove element i from set a
    srchbnd j;
    boolean found;
    found := false;  j := 1;
    repeat while j <= #a.size and not found do
      if #a.iset[j] = i then
        for k in [j+1...#a.size] do
          #a.iset[k-1] := #a.iset[k] od;
        #a.size := #a.size - 1;
        found := true;
      fi;
      j := j + 1;
    od;
  >>;

  has is <<
    formal r i; intset(r) a (varies) result boolean;
    $  set membership predicate
    srchbnd j;
    boolean res;
    res := false;  j := 1;
```

```
  repeat while j <= #a.size and not res do
    if #a.iset[j] = i then res := true fi;
    j := j + 1;
  od;
  return res
>>;

+ def <<                                                    7
  formal intset{r} a (copied); r i result intset{r};
  insert(a,i);
  return a
>>;

- def <<
  formal intset{r} a (copied); r i result intset{r};
  remove(a,i);
  return a
>>;

in def <<
  formal r i; intset{r} a  result boolean;
  return has(i,a)
>>;

empty def <<
  formal intset{r} a result boolean;
  $  true if a is empty, otherwise false
  [lbnd r...ubnd r+1] k;
  boolean res;
  k := lbnd r;  res := true;
  repeat
    if k in a then res := false fi;
    k := k + 1;
  until k > ubnd r or not res;
  return res
>>;

select def <<
  formal intset{r} a result r;
  $  select an arbitrary element from set a.
  $  assumes a is non-empty
  [lbnd r-1...ubnd r] k;
  integer m;
  k := lbnd r-1;
  repeat
    k := k + 1;
  until k in a ;
  return k
>>;

crintset def <<
  formal intset{r} z result intset{r};
  $  create an empty set                                   8
  return #↑(<false:i in r>::concrset)
>>;

opunion is <<
  formal intset{r} a (copied),
                   b (copied)
  result intset{r};
  $  union
  r k;
  intset{r} res;
  if empty(a) then res := b
  else
    k := select(a);
    res := opunion(a-k,b+k);
  fi;
  return res
>>;

+ def <<
  formal intset{r} a,b result intset{r};
  $  union of a and b
  return opunion(a,b);
>>;

symdiff is <<
  formal intset{r} a (copied),
                   b (copied)
  result intset{r};
  $  symmetric difference
  r k;
  intset{r} res;
  if empty(a) then res := b
  else
    k := select(a);
    if k in b then res := symdiff(a-k,b-k)
    else res := symdiff(a-k,b+k) fi;
  fi;
  return res
>>;

/ def <<
  formal intset{r} a,b result intset{r};
  $  symmetric difference of a and b
  return symdiff(a,b);
>>;

- def <<
  formal intset{r} a result intset{r};
  $  set complement with respect to r
  intset{r} b;
  b := crintset(b);
  for i in r do
    if not i in a then b := b + i fi;
  od;
  return b
>>;

% def <<
  formal intset{r} a (copied); r k result intset{r};
  $  form the set of all i+k, i in a.
  intset{r} z;
  r m;
  z := crintset(z);
  repeat until empty(a) do
    m := select(a);
    a := a - m;
    z := z + (m + k);
  od;
  return z
>>;

end;
          .
          .
          .

type connbnd is [0...5];
type connmatrix is array{intset{connbnd},connbnd};
          .
          .
          .

warshall is <<
  formal connmatrix a result connmatrix;
  $  warshall's algorithm for transitive closure
  for i in connbnd do
    for j in connbnd such i in a[j] do
      a[j] := a[i] + a[j]
    od;
  od;
  return a;
>>;
```

NOTES

1. The line containing the reserved word "space" is the header line for the definition of an "intset{r}" space, representing an abstract data type "set of integers." The identifier r represents a formal parameter which will take on a data type "value" when intset is used in a declaration. As the space is written this data type "value" must be a range data type [p...q]. An example of the declaration of a variable of "intset" data type is

```
intset{[1...100]} a;
```

Thus, the formal parameter r takes on the actual parameter value "[1...100]." The range r specifies the universe of values for the set.

2. "Concrset" is the shorthand notation for the concrete representation of an "intset". This representation involves a "record" of two items, a "size" representing the number of elements in the set and an "iset" representing the actual elements of the set as an array of integers.

3. Thus, the header line

```
space intset{r} def concrset;
```

specifies that an abstract type "intset" is being defined and that it is represented by the concrete type "concrset". Of course, the concrete type could have been another abstract type, with the restriction that some concrete type in a chain such as this must finally be represented by a data type which is built into the programming language.

4. In the type constant definition

```
type sizerng is
[0... ubnd r - lbnd r +1]
```

the operators "ubnd" and "lbnd" are compile-time operators which require a range data type [p...q]. The result of ubnd [p...q] is q. The result of lbnd [p...q] is p.

5. A procedure "insert" is defined as local to the space definition by virtue of the "is" following "insert" rather than "def". This procedure requires two input values: an "intset{r}" a and an integer i. The result returned is an "intset{r}". Note that actual parameters corresponding to "a" are passed by value, since records in the base language are passed by value under default circumstances.

6. The operator "#" appearing in the expression

```
#a.size
```

is termed a "concretion" operator. It has no effect at run-time. Its only effect is a compile-time type transfer of the identifier which it precedes. This is a transfer from the abstract type to the concrete type and is necessary to resolve ambiguities which might otherwise exist about the meaning of operators. Note that "a.size" is a type usage error since the dot (.) operator is defined only for records, not for intset{r}. However, "#a" transfers the type of "a" to that of "concrset" which is a "record."

7. The "def" following an operator (e.g., "+") or an identifier specifies that the body of that procedure is accessible outside the space as an operation. The occurrence of these operations outside the space result in a macro-like expansion of the procedure body following the "def". However, note that the "-" operation above contains nothing more than a procedure call on "remove". Thus, the expansion of the "-" body is into a procedure call on "remove"; "remove", since it is followed by "is" rather than "def" is a procedure local to and not accessible outside the space. It is implemented as a normal procedure and references to it from inside the space are implemented as procedure calls.

8. The operator #↑ is an "abstraction" operator which operates as the opposite of the "concretion" operator. Its function is to transfer the type of the expression following it from the concrete type to the abstract type.

9. The operators "#" and "#↑" may appear only with a space definition and may refer only to that data type being defined. These operators may not be combined to produce, for example, "##x". Thus, one may only transfer from the concrete to the abstract type, or vice-versa, of the data type being defined. Hence, all external access to an abstract data type is through the operators defined in the space with "def" following them. In particular, the programmer may not access (externally) the concrete form of the abstract type directly under any circumstances. He can, however, access it indirectly by providing an operator that will access it for him.

To illustrate this, suppose that it is required to know the number of elements in an intset{r}. While the size field cannot be referenced directly, an operator named "size" may certainly be provided. This does not violate the basic requirements of abstract data types since a "size" operator can always be reprogrammed, but a direct reference to a concrete structure makes assumptions about the structure which would require modifications in the abstract program if the concrete structure were modified.

III. Example 2 -- sets of integers over a range

This example illustrates what appears to be the same data type as in example 1. And, in fact, its external interface is precisely that of example 1. However, its internal, concrete structuring of sets of integers is quite different, in this case a more efficient structuring than that of example 1.

```
space intset{r} def concrset;
  type concrset is array{boolean,r};

  + def <<
    formal intset{r} a (copied); r i result intset{r};
    $  a + {i}
    #a[i] := true;
    return a
  >>;

  - def <<
    formal intset{r} a (copied); r i result intset{r};
    $  a - {i}
    #a[i] := false;
    return a
  >>;

  in def <<
    formal r i; intset{r} a result boolean;
    $  set membership operator
    return #a[i]
  >>;
```

```
        .
        .
        .
  (remainder of the space is unchanged)
        .
        .
        .
end;
```

The abstract program given previously for Warshall's algorithm is precisely the same as the abstract program given in example 1. The difference is the representation of integer sets in this example as bit strings rather than as integer elements within an array. The uniformity of the abstract program in these two examples illustrates one of the fundamental motivations for the use of abstract data types: A capability for changing the underlying representation of the elements of a data type without requiring a modification of the program that uses it.

IV. Example 3 -- an illustration of a subtle problem associated with type-checking in abstract data types

While all readers are no doubt aware of the mathematical concept of a complex number, there are many who may not be aware of a similar concept: a gaussian integer. Gaussian integers are restricted complex numbers having two pertinent characteristics:

(1) Gaussian integers are complex numbers whose real and imaginary parts are restricted to the integers.

(2) Addition, subtraction, multiplication and division are defined over the complex numbers; over gaussian integers, only addition, subtraction, and multiplication are defined.

This suggests that if we require both complex numbers and gaussian integers in an abstract program, we should define a space "arpair" as follows:

```
space arpair{t} def pair;
  type pair is record{a:t; b:t};

  + def <<
    formal arpair{t} x,y result arpair{t};
    $  x + y
    return #↑(<#x.a+#y.a, #x.b+#y.b>::pair);
  >>;

  - def <<
    formal arpair{t} x,y result arpair{t};
    $  x - y
    return #↑(<#x.a-#y.a, #x.b-#y.b>::pair);
  >>;

  * def <<
    formal arpair{t} x,y result arpair{t};
    $  x * y
    return #↑(<
      #x.a*#y.a-#x.b*#y.b , #x.a*#y.b+#x.b*#y.a
                          >::pair);
  >>;

  / def <<
    formal arpair{t} x,y result arpair{t};
    $  x / y
    return #↑(<
      (#x.a*#y.a + #x.b*#y.b)/(#y.a*#y.a + #y.b*#y.b),
      (#x.b*#y.a - #x.a*#y.b)/(#y.a*#y.a + #y.b*#y.b)
                          >::pair);
  >>;

  ↑ def <<
    formal arpair{t} x; integer y result arpair{t};
    $  x ↑ y
    arpair{t} z;
    z := mkpair(1,0);
    repeat while y > 0 do
      repeat until odd(y) do
        y := y div 2;
        x := x * x;
      od;
      y := y - 1;
      z := x * z;
    od;
    return z;
  >>;

  mkpair def <<
    formal t p,q; result arpair{t};
    $  form complex number or gaussian integer.
    return #↑(<p,q>::pair)
  >>;

end;
```

Now by simply defining the type constants

```
type complex is arpair{real}
type gaussinteger is arpair{integer}
```

we have a "complex" type and a "gaussinteger" type.

However, there is slightly more to be discussed about this example. If all space operator bodies are type-checked at the time the "arpair" space is activated (i.e., at the first occurrence of "arpair{real}" or "arpair{integer}") then the definition of the "/" operator will produce a type error in the activation "arpair{integer}." This is because the operator "/" always returns a real value in Model (as well as ALGOL 60 and Pascal). The solution to this anomaly lies in a decision to type-check abstract operator bodies only at the first occurrence of their use.

Thus, if "/" is never used over two gaussian integers, the body of "/" in "arpair{integer}" will never be type-checked and, hence, the type error will never occur.

V. Example 4 -- a vector data type over integers or reals

The same subtlety discussed in example 3 occurs in this example. However, it is resolved in a slightly different way.

Since vectors of both reals and integers will undoubtedly be required, both a "/" operator and a "div" operator are defined. As we learned above, this is no problem since type checking is done only at the first occurrence of a "/" or "div" operator, not when the space is "activated." Thus, if one uses the "/" operator with real vectors (or integer vectors) and the "div" operator only with integer operators then a type error will not occur as, of course, it shouldn't.

The primary reason for displaying this example is to show a good example of the three abstract data types that we have seen - intset, complex, and vector - interacting in a parallel version of the Fast Fourier Transform [6].

```
space vector[a,r] def vectype;
  type vectype is record[size:sizerng;
                         vec :array[a,bound]];
  type sizerng is [0...ubnd r - lbnd r + 1];
  type bound is [1...ubnd r - lbnd r + 1];

  + def <<
    formal vector[a,r] x,y result vector[a,r];
    $  x + y
    $  x and y are assumed to be of the same size
    return
      #↑[ <#x.size,
            <#x.vec[i] + #y.vec[i]:i in [1...#x.size]>
          >::vectype)
  >>;

  - def <<
    formal vector[a,r] x,y result vector[a,r];
    $  x - y
    $  x and y are assumed to be of the same size
    return
      #↑[ <#x.size,
            <#x.vec[i] - #y.vec[i]:i in [1...#x.size]>
          >::vectype)
  >>;

  * def <<
    formal vector[a,r] x,y result vector[a,r];
    $  x * y
    $  x and y are assumed to be of the same size
    return
      #↑[ <#x.size,
            <#x.vec[i] * #y.vec[i]:i in [1...#x.size]>
          >::vectype)
  >>;

  / def <<
    formal vector[a,r] x,y result vector[a,r];
    $  x / y
    $  x and y are assumed to be of the same size
    return
      #↑[ <#x.size,
            <#x.vec[i] / #y.vec[i]:i in [1...#x.size]>
          >::vectype)
  >>;

  / def <<
    formal vector[a,r] x; a y result vector[a,r];
    $  x / (scalar) y
    return
      #↑[ <#x.size,
            <#x.vec[i] / y:i in [1...#x.size]>
          >::vectype)
  >>;

  div def <<
    formal vector[a,r] x,y result vector[a,r];
    $  x div y
    $  x and y are assumed to be of the same size
    return
      #↑[ <#x.size,
            <#x.vec[i] div #y.vec[i]:i in [1...#x.size]>
          >::vectype)
  >>;

  subscript def <<
    formal vector[a,r] x (varies); r k result a;
    $  standard vector subscripting
    return #x.vec[k - lbnd r + 1]
  >>;

  subscript def <<
    formal vector[a,r] x (varies);
           intset[r] s (copied);
    result vector[a,r];
    $  form a new vector consisting of all x[i] such
    $  that i is in s.
    sizerng k;
    r m;
    vector[a,r] y;
    k := 0;
    repeat until empty(s) do
      m := select(s);
      s := s - m;
      #y.vec[k+1] := x[m];
      k := k + 1;
    od;
    #y.size := k;
    return y;
  >>;

  & def <<
    formal vector[a,r] x (copied),
                       y (copied)
    result vector[a,r];
    $  concatenate two vectors.
    $  the sum of the number of elements in vectors
    $  x and y is assumed to be less than or equal to
    $  the maximum number of elements allowed in a
    $  vector.
    for i in [#x.size+1...#y.size+#x.size] do
      #x.vec[i] := #y.vec[i-#x.size];
    od;
    #x.size := #x.size + #y.size;
    return x;
  >>;

  crvector def <<
    formal sizerng n result vector[a,r];
    $  create a vector of n elements
    return #↑(<n,<>>::vectype);
  >>;

  store def <<
    formal vector[a,r] x (varies); r i; a y noresult;
    $  subscripted vector assignment
    #x.vec[i - lbnd r + 1] := y;
  >>;

end;
```

The "vector" space demonstrates one new concept: the definition of two subscript operators over vectors. The definition of an operator of the form

subscript def <<...>>

will define a subscript operator in the space being defined. Outside the space the subscript form "x[i]" will designate the subscript operator.

One of the operators will probably require more explanation. It's function is similar to that of the APL "compress" operator. An example will suffice to explain the operation: Let A be the vector 14,12,6,-10,8,4,3,15,1,7 indexed [0...9] and let s be the integer set {1,4,7,8}. Then A[s] is the vector 12,8,15,1 The "&" operator concatenates two vectors.

VI. Example 5 -- a parallel version of the Fast Fourier Transform

```
        .
        .
        .
n is 8;
log2n is 3;
type vecrange is [0...n-1];
type complex is arpair{real};
type cmplxvector is vector{complex,vecrange};
        .
        .
        .
fft is <<
  formal
    [-1...1] sign;
    cmplxvector f (varies);
  noresult;
  $  discrete fourier transform of f (sign = -1)
  $  inverse discrete fourier transform of f (sign = +1)
  twopi is 6.283185308;
  cmplxvector abar;
  complex a;
  intset{vecrange} c,s,scomp;
  a := mkpair(cos(twopi*sign/n),sin(twopi*sign/n));
  abar := crvector(n div 2);
  for i in [0...n div 2 - 1] do abar[i] := a↑i od;
  c := crintset(c);
  for i in [0...n div 4 - 1] do c := c + 2*i od;
  s := crintset(s);
  for i in [0...n div 2 - 1] do s := s + i od;
  scomp := -s;
  for i in [1...log2n] do
    f := (f[s]+f[scomp]) & (abar*(f[s]-f[scomp]));
    if i < log2n then
      abar := abar[c] & abar[c];
      s := s / (s % 2↑(log2n-i-1));
      scomp := -s;
    fi;
  od;
  if sign = -1 then f := f / mkpair(n,0) fi;
>>;
```

VII. Abstract Data Types in Programming Languages - Basic Requirements

The following five characteristics are considered to be basic requirements for any data type abstraction mechanism in a programming language:

(1) Access to an abstract data type is allowed only through the operator set for the data type.

This requirement is the key to modularization, factorization and ease of modifiability. It is the foundation of the entire concept of abstract data types discussed in this paper. While external access to abstract types is analogous to side effects in procedures and functions, external type access must be strictly prohibited since it violates the basic principles so drastically. Side effects in procedures and functions are a much more controversial subject.

Thus, "complex" data elements may be accessed only through the operators "+", "-", "*", "/", "^" and "mkpair". It should be noted that the SIMULA class does not meet this requirement because local class variables can be accessed externally.

(2) Language constructs in the base language should be extendable to new abstract data types.

It is a relatively easy task to allow extension of operators to cover new data types. This seems an important thing to do, especially in the case of "complex" in which an arithmetic formula could become quite complicated; the availability of a functional notation alone would result in a LISP-like arithmetic confusion.

A more difficult task is to extend structure formers and certain control structures to include new data types. For example, suppose the language contains a for statement of the form

for i in S do L od

where i iterates over a set S of natural numbers. Then definition of a new data type of sets over real numbers should allow for the iterative form above to be extended to allow iteration over the new data type.

In a new data type "matrix," it is intolerable to lose the standard subscripting notation for matrices simply because subscripting in the base language cannot be extended to new data types.

(3) Definitions of abstract data types should allow for formal parameters in their definitions. Invocations through declarations of abstract data types should allow for corresponding actual parameters.

While this seems to be a debatable issue [14] to some, it is, nevertheless, still felt to be a requirement for languages supporting abstract data types. The example of "arpair" in example 3 should offer some convincing argument.

Another example might be the case of a new data type

list{elem}

that creates a linearly-linked "list" [7] of "elems". Here "elem" could conceivably include a host of different data types for different lists within a single program; likewise for queues, stacks, etc.

(4) The definition of an abstract data type should be implemented by the compiler as a truely new data type.

This topic illustrates an apparent misconception about abstract data types and stems from a question asked in [15]. Pascal does not provide for the definition of new (structured) data types. The

declaration

```
type candidate = array [1...m] of integer
```

does not imply that "candidate" is a new data type. This is because there is no distinction between "candidate" and "array [1...m] of integer " from the compiler's point of view. If a procedure requires a "candidate" and the programmer passes an "array [1...m] of integer", or vice-versa, this is perfectly legal and correct. Thus, "candidate" is no more than a shorthand abbreviation for "array [1...m] of integer in effect, the compiler can "see through" the identifier "candidate" to its definition "array [1...m] of integer".

The Model programming language supports the definition of truely new data types. Suppose Model spaces are declared with the forms:

```
space force,mass,acceleration def real;
  * def <<
    formal mass m; acceleration a result force;
    return #↑(#m*#a)
  >>;
  (other operations)
end;
```

The use of a real number or real variable where any of "acceleration", "mass" or "force" is required will result in a compiler diagnostic, since these terms are not merely abbreviations for real. In particular, if f is declared to be a "force" and r and s are declared to be merely real then

```
f := r * s;
```

will result in a compiler diagnostic, since "r * s" is real and f is a "force". It should be noted that in Pascal the type definitions

```
type force, mass, acceleration = real;
```

will not result in compiler diagnostics in the above case since these terms are not truely new data types. Thus, it can be said that in a language producing new data types, the compiler must not "look through" the new data type to its definition.

This is an important concept because the apparent evidence in type-checking languages is that the more data types known to the compiler the better diagnostics it can give about type errors.

(5) Operations on abstract data types should be implemented efficiently.

In some cases, operators associated with a data type will be nothing else than a reference into a record or an array when one of these structures is used to represent the abstract data type. For example, a space "matrix" will certainly have an operator for accessing an element of a matrix. Since the matrix will most likely be represented by a two-dimensional array, the matrix access operator will be translated into a two-dimensional array access: classical subscripting. To implement such an operator by subroutine calls is intolerable. In Model, all abstract data type operators are implemented as macro expansions producing inline code. Thus, the matrix access described above costs no more at run-time than a two-dimensional array access.

In cases where a macro call would cause too much code expansion, a mechanism exists for producing a subroutine call instead.

VIII. A Discussion Of Some Problems

There are several problems that the Model approach has not dealt with as yet.

(1) There is no device for restricting and checking allowable data types used as actual parameters in space activations.

For example, the "vector{a,r}" type is restricted to the formal parameter "a" taking on only data type values "real" or "integer". But there is no mechanism for specifying this and therefore the compiler cannot detect illegal space activations, such as

```
vector {boolean, [1...100]}
```

The existing type-checking in Model will detect this at a later time, but the error message will be "too late", a characteristic which should be avoided in compilers.

(2) Operators wihout formal parameters cannot be handled.

The Model compiler resolves operator ambiguities by operand type matching. Thus, for example, the meaning of "a+b" is resolved by noting the data types of a and b. However, if an operator has no operands then the ambiguity cannot be resolved by the compiler as it is currently implemented. The problem has been temporarily solved by providing a dummy operand so that the compiler can resolve the ambiguity and then ignore the operand within the operator body. Note the operator "crintset" in "intset{r}".

(3) The use of the abstraction operator (#↑) and the concretion operator (#) tends to make the space operation bodies appear cryptic and cluttered. It would seem a worthwhile task to enable the compiler to automatically determine type transfers when

possible; of course, when ambiguities occur that cannot be resolved then these operators will remain a necessity. It is easy to see that such ambiguities will occur when both the abstract type and the concrete type have the same operator defined over their respective spaces.

IX. Future Directions

The next phase of the Model project will be concerned with the inclusion of a capability for extending the Model iterative operators. This capability is considered to be of the utmost importance in the development of clean, readable programs. The extension of iterative operators takes two forms:

(1) the extension of the for statement and

(2) the extension of structure formers [10].

The inclusion of these two capabilities would greatly simplify the examples in this paper. Following are a few such simplifications:

The awkward "crintset" operator and its use as exemplified in the "fft" algorithm

```
s := crintset(s);
for i in [0...n div 2-1] do s := s+i od:
```

would be replaced by the more readable

```
s :=[) i:i in [0...n div 2-1] ():ndxset;
```

This states that s is the "ndxset" of all integers in the interval [0...n div 2-1]. The extension here is that of the creation operator which has the form "[) <iter> ():t" where <iter> is some iterator.

A more striking example is the "%" operator. The entire body could be replaced by

```
return [) i+k:i in a ();
```

Here the extension would involve the addition of a new iterative operator "in" to the intset{r} space.

Finally, we note that the "union" procedure body could be replaced by the more readable:

```
for k in a do b:=b+k;
return b;
```

While abstract data types have not been tested sufficiently in practice, the concept seems to have a bright future. Perhaps one of the greatest contributions may be in the area of dimensional analysis, a concept well known to the physical sciences. The example above defining "force","mass", and "acceleration" data types should illustrate how Model might be used to perform dimensional analysis. In performing type-checking over these data types, the compiler is, in actuality, performing dimensional analysis, by checking to see that the result of a multiplication by a "mass" and an "acceleration" is always used as a "force". Specifying such dimensional analysis relationships in a convenient manner is another effort which will be studied in the future as a by-product of abstract types in Model.

References

1. Dahl, O.-J., Myhrhaug, B., and Nygaard, K. The SIMULA 67 Common Base Language. Publication S-22, Norwegian Computing Center, Oslo, 1970.

2. Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. Structured Programming. Academic Press, New York, 1972.

3. Geschke, C.M., and Mitchell, J.G. On The Problem of Uniform References to Data Structures. Proceedings of International Conference on Reliable Software, Los Angeles, June, 1975, 31-42.

4. Hoare, C.A.R., Proof of Correctness of Data Representations. Acta Informatica 1 (1972), 271-281.

5. Hoare, C.A.R., Data Reliability. Proceedings of International Conference on Reliable Software, Los Angeles, June, 1975, 528-533.

6. Kahaner, D.K., and Morris, J.B. Madcap 6 and the Fast Fourier Transform. Los Alamos Scientific Laboratory, Los Alamos, New Mexico.

7. Knuth, D.E. The Art of Programming: Volume 1. Addison-Wesley, 1968.

8. Liskov, B. A Note on CLU. Memo 112, Computation Structures Group, Massachusetts Institute of Technology, Project MAC, Cambridge, November, 1974.

9. Liskov, B. and Zilles, S. Programming with Abstract Data Types. Proceedings of SIGPLAN Symposium on Very High Level Languages, Santa Monica, 1974, 50-59.

10. Morris, J.B., Structure-Formers in Programming Languages. Los Alamos Scientific Laboratory, Los Alamos, New Mexico, September, 1975.

11. Morris, J.H., Types are not Sets. Proceedings of SIGPLAN/SIGACT Symposium on Programming Languages Boston, 1973, 120-124.

12. Morris, J.H. Protection in Programming Languages. Comm ACM 16, 1(Jan, 1973).

13. Parnas, D.L., Information Distribution Aspects of Design Methodology. Proceedings of the IFIP Congress, August, 1971.

14. Wasserman, A.I.(Ed) SIGPLAN NOTICES 10, 7 (July, 1975), Question by Organick, 31.

15. _____, Question by Ison, 31.

16. Wells,, M.B., A Data Type Encapsulation Scheme Utilizing Base Language Operators. Submitted to this conference.

17. Wirth, N. The Programming Language Pascal. Acta Informatica 1,1 (1971), 35-63.

18. Wulf, W.A. Alphard: Toward a Language to Support Structured Programs. Dept. of Computer Science Internal Report, Carnegia-Mellon University, Pittsburgh, April, 1974.