A DATA TYPE ENCAPSULATION SCHEME UTILIZING BASE LANGUAGE OPERATORS*

By

Mark B. Wells

and

Fred L. Cornwell

Abstract:

A data type encapsulation scheme in which the "space" operations are expressed naturally in terms of the base language operators is described. The scheme results from a conceptual separation of operators and procedure calls in the base language and produces a language of considerable expressive power. The scheme has been implemented and several examples are given.

1. Introduction

The data type encapsulation concept broached by Dahl, Myhrhaug, and Nygaard [1] and pursued by Hoare [2], Liskov and Zilles [6], and others [e.g. 3,8,12] is one of the more promising ideas of modern programming language design. By using data structure abstractions afforded by this concept, the user may naturally structure his programs into readable, easily modifiable and independent units which to a large extent can be verified by a type checking compiler. In a very real sense, complex data can now be synthesized from simpler structures in much the same way as complex algorithms can today be synthesized from simpler procedures.

This paper discusses the data type encapsulation scheme which has been welded onto the existing Madcap 6 programming language [7,10]. The primary goal has been to obtain the expressive power afforded by the encapsulation concept without unduly altering the extant language and compiler. This constraint of smooth language evolution has molded the scheme in interesting ways. Foremost in this respect is the restriction to base language operations on elements of a user defined "space" (Simula 67 [1] uses the term "class" and CLU [6] the term "cluster" for such an encapsulated abstract data type). This restriction is a result of the treatment of procedures as a base language data type--procedure call is denoted by the juxtaposition (an operator!) of a procedure variable and a list of actual parameters. While there is some loss of generality in this scheme, there are compensating benefits. One is that it was very easy to design the language to allow the natural operator notation, i.e. "a+b" instead of "plus(a,b)", for expressing operations on elements of a space. Another is that it is possible for the user to define new operations on functions using natural notation--composition by "f ∘ g" for example. Due to the very large number of base language operators (seventy-two), these expressive capabilities compensate, in our opinion, for the lack of an indefinite list of operators for use in the abstract types.

Besides the function (≡ procedure) data type, there are a number of other Madcap features which significantly impinge

*This work was supported by the Division of Physical Research-Molecular Sciences of the United States Energy Research and Development Administration.

upon the encapsulation scheme. We do not intend to describe these features nor the encapsulation scheme itself in formal detail. By means of a number of examples, we hope to illustrate the expressiveness and power of extensible Madcap as it exists today. This work shows how truly significant gains in algorithm expressibility are possible by relatively few syntactic additions to the language. Only one new data type, SPACE, with its attendant value form, and two new operators were necessary to implement the concept. We believe such a language upgrading would not be difficult for any other existing language which already possesses certain basic characteristics such as hierarchial data structuring facilities and block structuring with activation record retention.

2. Basics

Madcap is a block structured language in which the primary blocks are function definitions, that is, constant values of function data type. A variable is declared to be local to a function by virtue of being underlined where it is first assigned a value. The data type of the variable is assumed from the type of the value to which it is being assigned. There are base language constructs which produce actual values for the base language types; for example,

```
3.75
```

is a constant of type REAL, while

```
«x:real; 3x^3-4x^2+5»
```

is a value of type FUNCTION which accepts a single real argument and returns a real result. In addition, there are base language constructs, namely real, boolean, etc., which represent undefined values of the described type. Thus, for example, the assignment

```
r ← real
```

is a declaration of r as a local real variable. Of course the statement

```
r ← 3.75
```

accomplishes both declaration and initialization. Types propagate through base language operators in a natural fashion (see Section 7), so that the right-hand side of a "declarative assignment" need not be a constant; for example, if r were a real variable then the assignment

```
s ← r^2+1
```

declares s to be real also, as well as assigning to it the indicated value.

The primitive base language types are REAL, BOOLEAN, and STRING. The types SEQUENCE, STRUCTURE (essentially a RECORD of Pascal [11]), SET, and FUNCTION are non-primitive in the sense that the precise type definition depends on the types of certain component parts [10]. For instance, a structure

```
S ← (↓tag:boolean; ↓data:real)
```

is a pair of values, the first of which is a boolean value and will be referenced by the field name tag (e.g. by S↓tag or by tag↑S) and the second of which is a real number and will be referenced by the field name data. A list of the base language types and the syntactic representation of their values is given in Table I.

TABLE I

Base Language Types and Their Value Forms

Data Types	Value Forms (Examples)
REAL	7.25, 163, -14.8, real
BOOLEAN	true, false, boolean
STRING	"dog", "A_i", "Value="
FUNCTION	«x:real; x^2-3»
SET	{1,4,7,8}, {"A", "B", "C"}
SEQUENCE	(18.4, 7.5), (true, false, false)
STRUCTURE	(↓tag:true; ↓data:33.7)
SPACE	$(real; ↓pls:«...»; ↓mns:«...»)$

The type SPACE is the new type that permits user defined abstract types. A particular space value consists of a representation for elements of the space and the list of operations allowed on those elements all bracketed by $< ... >$. The operations, which are some subset of those for which a base language operator exists, are given as functions. They are associated with the corresponding operator by means of a mnemonic field name. This operator/field name correspondence is given in Table II and is known to the compiler. [This mnemonic notation is as distasteful to the authors as it must be to the readers. However, its use was extremely expedient. Future alternatives are discussed in Section 7.] Even though an operator is given mnemonically in the specification of the space, when invoked, it is indicated symbolically. A simple example of a space is the following partial definition of a complex data type:

```
COMPLEX ← $(
 (↓Re:real; ↓Im:real);
 ↓mns:«
   (x,y):@COMPLEX;
   (Re↑x`-Re↑y`, Im↑x`-Im↑y`)@COMPLEX
  »;
 ↓eql:«
   (x,y):@COMPLEX;
   Re↑(x-y)`=0 ∧ Im↑(x-y)`=0
  »;
 .
 .
 .
)$
```

TABLE II

Base Language Operators and Associated Mnemonics

Op	Mn	Op	Mn	Op	Mn	Op	Mn
|x|	mag	∴x	thr	x+x	pls	x=x	eql
||x||	nrm	¬x	not	x-x	mns	x≠x	neq
⌊x⌋	flr	∫x	ing	x x	jux	x<x	les*
⌈x⌉	clg	∇x	del	x·x	bye	x>x	gtr
$\binom{x}{x}$	bno	∂x	par	x×x	tms	x≤x	lse*
		⊳x	ahd	x/x	ovr	x≥x	gte
$\{{}^{x}_{x}\}$	stn	†x	exe	x÷x	rdv	x∈x	elm*
		@x	pat	x mod x	mdd	x∉x	nel*
x_x	sub	Σx	sum	x⊛x	cpl	x⊂x	sbs*
x^x	pwr	Πx	prd	x⊖x	cmn	x⊄x	nsb*
		MAX x	max	x∪x	jon	x⊃x	sst
$\frac{x}{x}$	ddv	MIN x	min	x∩x	met	x⊅x	nsp
+x	prp	Ux	unn	x~x	smn	x⇒x	daf
-x	prm	Πx	int	x∘x	bdt	x⇐x	dab*
~x	prs	∃x	exs	x&x	amp	x≡x	idn
√x	sqt	∀x	all	x@x	att*	x≈x	aeq
#x	car	ANY x	any	x:=x	ceq*	x≅x	isv
@x	cop	x!	fcl	x∨x	orr	x±x	pmn
		x%	pct	x∧x	ann	x∓x	mnp

Notes: 1) x is a placeholder.
2) An * indicates that the type derives from the right operand; for all others, the type derives from the left operand.

As indicated by the first component of the space, a complex number is represented by the structure which is a pair of real numbers respectively referenced by the field names Re and Im. For simplicity of illustration, we have only given operations corresponding to the operators -(minus) and =(equals). They are sufficient, however, to illustrate the relevant notations associated with spaces. The @(at) infix operator, read "from", determines the space (right operand) of a computed value (left operand) as in the result of the minus function. When there is no left operand, that is, if @ is a prefix operator, then reference is being made to an undefined element from the space, as in the declaration of the formal parameters for both functions. The `(accent) suffix operator, read "representation", indicates that the operand should be treated with the type of its representation rather than its own type. For instance, in the minus function, the formal parameters x and y are declared to be variables of type COMPLEX. In order to implement this COMPLEX subtraction it is necessary to reference the real and imaginary parts of the representations of x and y (e.g. Re↑x`, etc.). By contrast, note that in the equals function the representation operator is applied after the meaningful minus operator has been applied to the two complex numbers.

The @ operator is also used external to a space to declare the type of variables. For instance, the statement

```
X ← (0,0) @ COMPLEX
```

declares X to be of type COMPLEX and initializes it to the zero complex number. The ` operator may also be used external to a space, with no change in its meaning.

To further familiarize the reader with basic Madcap notation, we present the following vector space example. With the space variable VECTOR defined as

```
VECTOR ← $(
  (real: 3 items);
  ↓pls: «(u,v):@VECTOR;
    (u`ᵢ+v`ᵢ: 0≤i<3) @ VECTOR»;
  ↓mns: «(u,v):@VECTOR;
    (u`ᵢ-v`ᵢ: 0≤i<3) @ VECTOR»;
  ↓bye: «u:@VECTOR; c:real;
    (u`ᵢ·c: 0≤i<3) @ VECTOR»;
  ↓eql: «(u,v):@VECTOR;
     (∀0≤i<3 u`ᵢ=v`ᵢ)»
)$
```

and the vector variables v and w defined as

```
v ← (5, -1, 4) @ VECTOR
```

and

```
w ← (-7, 8, 4) @ VECTOR
```

the sum v+w would equal < -2, 7, 8 > and the equality test v=w would return false.

3. Procedures

Only operators defined in the syntax of the base language have an associated mnemonic field name for use within a space definition. However, any other identifier can be used as a field name within a space and can be referenced in the same way as field names of structures. That is, a procedure exp could be defined within a space,

```
TYPE ← $(
  .
  .
  .
  ↓exp: « ... »;
  .
  .
  .
)$
```

for example, and then referenced as

```
exp↑TYPE(x)
```

for instance. To obviate the required repeated use of "↑TYPE" the user can take advantage of the function data type of the language and write

```
exp ← exp↑TYPE
```

at the beginning of the block (function) for which that definition applies; references would then become

```
exp(x)
```

simply. A similar assignment technique, e.g.

```
binomial ← bno↑TYPE
```

could be employed when the user would rather write procedure calls than use operator symbolism.

Another use of a procedure defined as a field of a space is for incorporating a create operation. For instance, one could have

```
↓create:«(a,b):real; (a,b)@COMPLEX»
```

within the COMPLEX space defined above. A user could then define a complex number with a normal procedure call, e.g.

```
X ← create↑COMPLEX(0,0)
```

instead of being required to know the complex number representation that is defined within the space, as in the example given previously:

```
X ← (0,0) @ COMPLEX
```

Of course, procedures operating on space elements can always be defined independent of the space definition itself, in which case their reference is subject to the normal type checking rules.

4. Base Language Spaces

Not all of the operations defined on the base language data types are effected with in-line code produced by the compiler. Certain involved operations, e.g. !(factorial), have always been implemented with procedures written in Madcap. With the space concept, this is accomplished very naturally. For instance, the spaces REAL, BOOLEAN, STRING, SEQUENCE, STRUCTURE, SET, and FUNCTION, and even SPACE itself are defined in the environment of everybody's programs. When encountering an operator and a type, the compiler first asks if an in-line operation is available. If not, then a scan through the appropriate space is made and if a match occurs, the associated function call replaces the operator. If no match occurs, the compiler reports a semantic error. This same process applies to user defined types as well, only in-line codes are never available.

The representation of elements of primitive base language spaces is expressed in terms of the lower case keywords real, boolean, etc. which mean an undefined value of the corresponding type. Notationally, therefore,

```
real ≡ @REAL
boolean ≡ @BOOLEAN
```

etc. for purposes of type declaration.

For an example, the space REAL is partially defined below:

```
REAL ← $(
 real;
 .
 .
 .
 ↓fcl: «
   n: real;
   if n<0 ∨ n>12 ∨ n≠⌊n⌋:
         error(!--bad parameter)
   else: if n=0:
         1
   else:
         n·(n-1)!
  »;
 .
 .
 .
 ↓pwr: « ... »;
 .
 .
 .
 ↓pls: «(x,y):real; x+y»;
 .
 .
 .
$)
```

The factorial (fcl) routine is given in its entirety. Note the use of the exclamation point to invoke a recursive call of the operation; recall that the compiler has Table II in it. Exponentiation using the natural superscript notation, e.g.

$$\ldots \quad x^{y} \quad \ldots$$

is available in Madcap, and its use, for x real, results in a call to the pwr routine of the space REAL. Finally, we have shown a plus (pls) routine in this space. While the + operator applied to real numbers does result in in-line code with the existing compiler, this routine would be needed if this space were just one of several values of a space variable--see Section 5.

The only operations on functions (besides assignment) which produce in-line code are evaluations. When the function has parameters, evaluation is indicated by the juxtaposition operator, e.g. sin X, f(x+1), proc(a,b). [Juxtaposition, when it is not used for identifier formation which takes precedence, is a perfectly good operator of the language which has meaning when applied to other types besides functions, e.g. reals.] When the procedure has no parameters, evaluation is indicated with the † (dagger) prefix operator, e.g. †$. The space concept allows other operations to be defined naturally, as illustrated below.

```
FUNCTION ← $(
 «x:real; real»;
 ↓bdt:«
   (f,g): «x:real; real»;
   «x:real; f(g(x))»
  »
)$
```

This is a user defined base language space restricted to one parameter real functions. A composition operator (∘ ≡ bdt) has been defined on the elements. Because its name is FUNCTION, it is a base language space and operations on functions within the scope of its definition which do not produce in-line code will be referred to it. If p and q were functions, e.g.

```
p ← «x:real; 3x^2-x-37»

q ← «x:real; -x^3+4»
```

then one could form their composition by using the ∘ (big dot) infix operator, e.g.

```
r ← p∘q
```

The value of r, namely the output of the composition operation, is again a function. The implementation of this feature requires that the language have an associated

computational model with activation record retention since f and g are global parameters of any value produced by this operation. In Madcap, this is accomplished using an implementation of Johnston's contour model [4]. One could now evaluate r. For example,

```
... r(2) ...
```

would have the value 15.

5. Space Forming Programs

The encapsulated data type concept has been implemented here by introducing a new data type, that of spaces, into the base language. This means that there are values of type SPACE and that there can be space variables which receive their values through the normal mechanisms of assignment and parameter passing. There can even be (and, in fact, is) a base language space named SPACE, although it is somewhat mind boggling to reflect on various operations of this space. Nevertheless, one of the true virtues of the space concept is that it is now possible to piece together abstract data types to form involved structures with little or no alteration necessary within the pieces.

A user can form composite spaces, for example multiply precise complex numbers, matrices of complex numbers, or matrices of multiply precise complex numbers, etc., with little more effort than is required to construct the basic spaces. One way would be to replace, by editing or rewriting, each reference to an element of the underlying type by the appropriate reference to an element of the new underlying type. For instance, in a PRECISE.COMPLEX space the representation might be

```
(↓Re:@PRECISE; ↓Im:@PRECISE)  .
```

The real of the COMPLEX example given in Section 2 has been replaced by @PRECISE where PRECISE is a space which effects multiple-precision arithmetic.

A more general technique, however, would be to write a procedure that has the underlying space--PRECISE, REAL, or what-have-you--as input parameter and returns the composite space as output.

For illustration, we show how a vector space over an arbitrary ring can be formed--this example can be given in its entirety here. In the vector space example of Section 2, the vector components were specifically real numbers; we now write a procedure which uses an input parameter to determine the type of the components, that is, to determine the base ring of the vector space. The operations in the ring are (say) add, subtract, multiply, and test for equality. We therefore first define a generic ring,

```
RING ← $(
 general;
 ↓pls: «(a,b):general; @RING»;
 ↓mns: «(a,b):general; @RING»;
 ↓bye: «(a,b):general; @RING»;
 ↓eql: «(a,b):general; boolean»
)$
```

for example. [We use the word general to indicate an undefined value of arbitrary type.] We then can write the following vector forming procedure which has the operations add(+), subtract(-), scalar multiply (•) and equals(=).

```
VECTOR.FORMER ← «
r:RING; n:real;
V ← $(
 (@r: n items);
 ↓pls: «
   (u,v):@V;
   (u`ᵢ+v`ᵢ: 0≤i<n) @ V
  »;
 ↓mns: «
   (u,v):@V;
   (u`ᵢ-v`ᵢ: 0≤i<n) @ V
  »;
 ↓bye: «
   u:@V; c:@r;
   (u`ᵢ·c: 0≤i<n) @ V
  »;
 ↓eql: «
   (u,v):@V;
   ∀ 0≤i<n u`ᵢ=v`ᵢ
  »
)$
»
```

This procedure has two formal parameters, r, which is the same type as RING, and n, which is a real number (i.e. of the same type as real). The local variable V is used merely to give a name, for reference purposes, to the space being formed as the output value of the procedure.

A user can then form a specific vector space merely by calling VECTOR.FORMER with two arguments, a specific ring and the number of components. For example, with a ring of integers (having the same form as RING) defined as

```
R ← $(
 real;
 ↓pls: «(a,b):@R; (a`+b`)@R»;
 ↓mns: «(a,b):@R; (a`-b`)@R»;
 ↓bye: «(a,b):@R; (a`·b`)@R»;
 ↓eql: «(a,b):@R; a`=b`»
)$
```

a vector space VECTOR would be formed by the assignment

```
VECTOR ← VECTOR.FORMER(R,5)
```

Variables of this new type VECTOR can now be formed and manipulated upon using reasonably natural notation, for example.

```
(u,v) ← @VECTOR
c ← @R
.
.
.
z ← u+v·c
```

The important idea here is that the VECTOR.FORMER procedure can be used without alteration to form vector spaces over other rings. More generally, any procedure can now be written with the type of its data as an input parameter. This permits the symbolism of the algorithm to be independent of the data on which it operates.

6. An Application

Thus far, we have concentrated on the definition and formation of spaces themselves. We now give an example that illustrates the expressibility afforded by use of the space concept. This example is taken directly from a running program [5].

The problem, which need not be understood to appreciate the notation, is to calculate one of the weights associated with a d th degree Newton-Cotes quadrature formula in n dimensions. The input to the procedure is a composition of 2^d into n+1 parts, and the output is a real number weight associated with that composition. The procedure is an implementation of a method due to Silvester [9]. As far as we are concerned here, it is just a manipulation of polynomials in n+1 variables. In fact, the relevant space is the space of multivariate polynomials.

Let $C=\langle c_0, c_1, \ldots, c_n \rangle$ be the input composition; for example, if n=2 and d=2, then <1, 2, 1> would be a possible composition. The method first forms

$$Q = \Pi_{0 \le i \le n} R_{c_i}(z_i)$$

where

$$R_m(z) = \frac{1}{m!} \Pi_{0 \le k < m} (d \cdot z - k) \ .$$

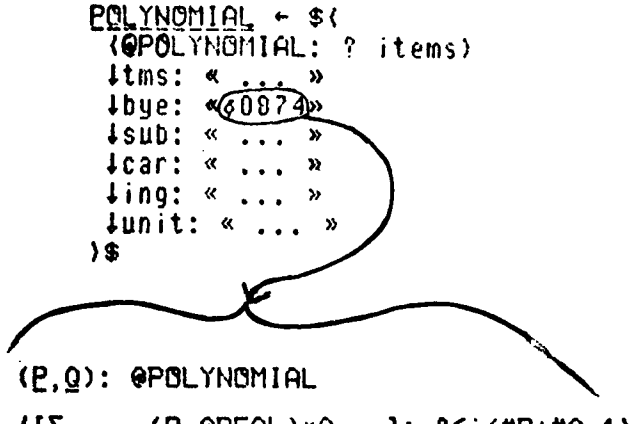
Each R is a polynomial in a distinct variable, z, and Q is a polynomial in n+1 distinct independent variables, z_i. The output weight is then n! times the integral of Q over the n dimensional "unit simplex" (a simple triangle for n=2).

Given the appropriate operations on polynomials, the program that implements this method is

```
«
C: Composition
n ← #C-1
d ← Σ_{m∈C} m

Q ← unit↑POLYNOMIAL(0)

for m ∈ C:
     R ← unit↑POLYNOMIAL(1)
     for 0 ≤ k < m:
          R ← R·(-k, d)
     Q ← Q×R

n!·∫Q·Π_{m∈C} 1/m!
»
```

This program utilizes three operators, •(≡ bye), × (≡ tms) and ∫ (≡ ing), and the unit creation procedure from the space of polynomials. The unit creation procedure has as its argument the number of variables of the polynomial being formed. The • operation multiplies two univariate polynomials and produces a univariate polynomial. It is used in the formation of the R polynomials. Note that a polynomial is represented by a sequence of its coefficients, in the case of a univariate polynomial by a sequence of reals. The × operation multiplies a polynomial in s variables by a polynomial in t different variables to produce a polynomial in s+t variables. Finally the ∫ operation performs the integration over the appropriate simplex.

For reasons of brevity, we do not include here a complete description of the POLYNOMIAL space used in this application. However, below we do give the top level of its definition and the definition of the • operation. Note that the space contains operations, e.g. subscript, that are not used in the above application but that are used within operation procedures of the space.



7. Discussion

The examples given in the previous sections indicate the data type encapsulation scheme of Madcap as it exists now (January, 1976). While we do not yet have abundant experience, we can make a few observations.

The conceptual separation of base language operators and program procedures is a fundamental characteristic of the language. Nevertheless, we believe the large operator vocabulary will make ours a useful scheme. We will possibly add a few more very common mnemonic operators such as **ln, sin, arctan, max, read, write** to the base language operator list.

The mnemonic field names are used at the present for the sake of expediency due to properties of earlier implementations. When time permits we will consider using the operators themselves whenever possible, as in

```
↓+ :« ... »  for  ↓pls:« ... »
↓LL:« ... »  for  ↓flr:« ... »
```

In cases where it is not possible we will perhaps utilize a placeholder scheme such as

```
↓θ^θ:« ... »  for  ↓pwr:« ... »
↓θθ:« ... »   for  ↓jux:« ... »
```

At the present time, type checking of operands and parameters by the Madcap compiler is minimal. Improved type checking would certainly aid the programmer by preventing him from compiling inexact programs. However, in our opinion, the expressibility and power of the language, which permit and promote good programming habits, is of foremost importance. Thus, we have concentrated on the language with only little regard to what effect future compiler improvements might have. Along a similar vein, we are relying, perhaps naively, on the availability of the powerful and convenient language constructs to prevent abuse of spaces. At this time we are not disallowing, a la CLU for instance, "abnormal" external access to space components.

It should be noted that in our scheme type checking is independent of type propagation. For the sake of simplicity, we propagate types from a single operand; for instance, the type for + derives from the type of the left operand while for < the type derives from the right operand (see Table II). This derived type is then used to determine the in-line code or space function call associated with that operator. Type checking would then determine if the types of the other operands (or parameters) properly match. While simple, our type propagation scheme nevertheless does have problems; for instance, because multiplication propagates the type of the left operand, the scalar multiplication of a vector space (see example of Section 5) must be written with the scalar on the right! Perhaps a slightly more elaborate type propagation scheme is desirable; although, in our opinion, it is going too far to check for an exact match between all operand types and the space operation parameter types. It would seem that it should be easy for the user, as well as the compiler, to determine the space to which an operator with arbitrarily mixed operand types belongs.

There are a few base language operators which, at least for the present, cannot be used within spaces or are otherwise special:

(1) The ↑ and ↓ (field referencing operators) are not allowed within spaces. This is to avoid massive changes in the evolving compiler.

(2) The ← (arrow assignment) operator is not allowed in spaces. We decided to go with this restriction because the base language assignment operation cannot be replaced by a procedure call due to the fact that there is no reference or pointer type per se in the language [10]. However, the :=(Algol assignment) operator is available and can be used to accomplish a copy-like assignment operation, which by the way is not equivalent to arrow assignment; for example,

```
↓ceq:«
  (a,b):@COMPLEX;
  Re↑a` ← Re↑b`; Im↑a` ← Im↑b`
 »
```

might be such a definition in the complex space.

(3) The subscript operator is allowed within spaces but since all available operators are access operators only it is not meaningful to use that operator on the left of an arrow assignment even though it is syntactically legal at the present time. For instance, a subscript operation on real vectors could be defined by

```
↓sub:«U:@VECTOR; i:real; U`i»
```

but only used on the right of an assignment

```
A ← (7,3,8) @ VECTOR
x ← Ai+2 .
```

(4) The (representation) operator cannot be defined as a space operator since it already has meaning for an operand of any type.

(5) The @(from) operator has base language semantics only when the right operand is a space. Thus, it is available for values of other types.

(6) The extension of control structure operations such as "if B:A", "for s∈S:", and "for x≤i≤y" where B,S, and x are elements of user defined spaces is important but still in the future for Madcap. The complete list of available operators appears in Table II.

The subjects of discriminated union, recursive data structures and run-time type checking have not been discussed in this paper. Certain aspects of these concepts do exist in the Madcap language today, but lack of time and experience have prevented a proper unification with the encapsulation scheme described here.

Our final comments concern efficiency. Application of the tools espoused here would appear at first glance to involve significant inefficiencies of computer usage. To a certain extent that is true, but it is extremely nearsighted to be overly bothered by these inefficiencies. The application of unifying tools such as these can allow significant reduction of development costs for involved calculations as well as reduction of computer execution time through "global" optimizations. Moreover, because of our better understanding of programming techniques we are now on the verge of mechanizing certain "local" optimizations. For example, it is quite easy for a Madcap compiler to determine whether or not a variable really varies. Thus, operations of a truly constant space could be inserted as in-line code (macros) rather than as expensive procedure calls. [Note that this is possible without conscious help from the user, that is, without having the language cluttered up with macro definitional facilities or the like.] In our opinion, the benefits from the expressive capabilities of the language and the potential for diagnostic guidance and local optimization from the compiler truly make the data type encapsulation concept a major advance in programming.

BIBLIOGRAPHY

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

2. Hoare, C. A. R. "Proof of Correctness of Data Representations", Acta Informatica, Vol. 1 (1971/1972), pp. 271-281.

3. Johnson, R. T. and Morris, J. B. "Abstract Data Types in the Model Programming Language", Proceedings of ACM Conference on Data: Abstraction, Definition, and Structure (this Volume), Salt Lake City, March, 1976.

4. Johnston, J. B., "The Contour Model of Block Structured Processes", Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices, Vol. 6, no. 2 (February 1971), pp. 55-82.

5. Kahaner, D. K. and Wells, M. B., "N-Dimensional Adaptive Quadrature Using Simplicial Subdivision", in preparation.

6. Liskov, B. and Zilles, S., "Programming With Abstract Data Types," Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN notices, Vol. 9, no. 4 (April 1974), pp. 50-59.

7. Morris, J. B. and Wells, M. B. "The Specification of Program Flow in Madcap 6," SIGPLAN Notices, Vol. 7, no. 11 (November 1972), pp. 28-35.

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

9. Silvester, P., "Symmetric Quadrature Formulae for Simplexes", Mathematics of Computation, Vol. 24, no. 109, (January 1970), pp. 95-100.

10. Wells, M. B. and Morris, J. B., "The Unified Data Structure Capability in Madcap 6," Inter. J. of Computer and Information Sciences, Vol. 1, no. 3 (September 1972), pp. 193-208.

11. Wirth, N. "The Programming Language PASCAL", Acta Informatica, Vol. 1, no. 1 (1971), pp. 35-63.

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