Implementation and application of a function data type*

by MARK B. WELLS

Los Alamos
Scientific Laboratory
of the University of California
Los Alamos, New Mexico

ABSTRACT

The modularization construct of subroutine, function, or procedure is well established in the scientific programming languages of today. In most cases, however, the construct is static in that once a function is defined and named it remains unchanged throughout the scope of its definition. We are concerned in this paper with the generalization to a function data type, that is, to the situation where one may have variables of type function which assume different specific procedures as their value within their scope. Notationally, implementation of the concept is made feasible by use of a juxtaposition operator. Computationally, the concept derives its usefulness from use of the contour model which allows activation record retention. Examples are given which show that the function data type concept subsumes less general techniques such as "coroutines" and "stream functions."

INTRODUCTION

Certain concepts of high-level, scientific programming languages such as recursive procedures, block structuring, and the general while and if-then-else control structures are well accepted due to their generality and usefulness in writing readable, well-structured programs. Other concepts such as hierarchical data structuring and abstract data types, while of more recent vintage, are fast proving their worth in the context of high-level languages. In this paper, we discuss experience with a function data type, a concept which we believe has promise of becoming as important as those mentioned above. This belief is based on the fact that function variables significantly assist with the high-level implementation of "generation procedures" and "backtracking" and a number of other less general organizational approaches such as "coroutines." Furthermore, our implementation of the function data type helps resolve the "harmful global variable" problem.4

The particular programming language involved in this work is Madcap, for which documentation is fragmented and incomplete.5 However, this paper is self-contained since all relevant notation is either self-explanatory or explained herein.

NOTATION AND IMPLEMENTATION

Data of a Madcap program are categorized into objects called "spaces." A space includes a description of the operations that can be applied to the associated data items and, if the space is defined by the user, may include a description of the representation of a typical item.6 There are certain spaces, called base language spaces, which have a syntactic construct for constants from that space. For example, REAL is a base language space with constants written 5.15, −623.9, 17, ... This space contains various operations—add, divide, square root, etc.—some being predefined by the particular computer system and others being defined at the language level. Terminologically, we say "a variable is of type real" or "has data type real" when the values it may assume and the primary operations that may be applied to it are from the space REAL. The base language spaces and examples of their constant form are given in Figure 1.

Note with regard to functions that the formal parameters are listed first using a colon notation and that the value of the last evaluated expression is returned as the result of a function's evaluation.

Madcap is a block structured language in which the text of a function definition, i.e., a constant of type function, is a block. Identifiers are local to a block if they are underlined at their point of definition within the block; they are global otherwise. Since identifiers are used solely as names of variables and there are no explicit declaration statements in the language, this point of definition will either be on the left of an assignment statement as in

\[ y \leftarrow 7.1 \]
or in the range of a loop statement (or loop expression) as in

\[
\textbf{for } 0 \leq j < I : \quad A_i = 2 - 1
\]

The data type of an underlined variable is derived from context, in the case of an assignment from the type of the expression on the right of the arrow. For instance, the assignment statement above serves to declare \( v \) to be a local variable of type real as well as indicating that when executed the value 7.1 is to be assigned to the variable \( v \).

There exist well-defined rules of type propagation through expressions; and since there are no jump statements in the language, it is possible to compute the types of all variables at compile-time by a direct flow analysis.7 (Of course, the program must be complete, including all referenced spaces, and must be properly ordered, with all variables defined before they are used. This last requirement does not preclude the use of recursive functions and data which are an integral part of the Madcap language.) The precise operation that is being specified by an expression depends on the types of the operands as well as on the operator. For instance,

\[
x + y
\]

indicates a real addition if \( x \) and \( y \) are of type real while it indicates a matrix addition if the operands are of type matrix. In the latter case, a function for actually performing the matrix addition would exist in the (user-defined) space named \texttt{MATRIX} and would be called upon to perform the operation.

There are about ninety operators in the Madcap language for which operations can be defined in various spaces.8 Some of these are predefined for base language spaces. For instance, \(+, -, \times, \div, \sqrt{}\), etc., imply established operations in the space \texttt{REAL}, while \( ^*, \vee \), and \( \wedge \) specify base language operations available on boolean variables. The only predefined operation of the \texttt{FUNCTION} space is evaluation. If the function value has no arguments then this operation is specified by the dagger (\(\dag\)) prefix operator, for example

\[
\dag \text{proc.}
\]

If the function has arguments, the evaluation is usually specified by the juxtaposition operator (one of the ninety), for example

\[
f(x+1) \quad g(x, y) \quad \text{exists } x \quad f(x) \quad (f)x.
\]

Since juxtaposition for identifier formation takes precedence over juxtaposition as an operator, it is not possible to have the immediate juxtaposition of two letters of the same case imply function evaluation (or any other operation for that matter). However, this is not serious since spaces separate tokens as in the third example above, and the dot (\(\cdot\)) infix operator also specifies function evaluation as in

\[
f \cdot x \quad g(x, y) \quad f(a) \cdot b
\]

for example. (The juxtaposition operator has precedence over the dot operation.)

The contour computational model\(^9\) is used by Madcap. This means that the environment of a function value (block) is retained if needed for use by all activations of that value. Therefore, a function value must be represented by a pair of pointers, a pointer to the code associated with the value (code pointer) and a pointer to the activation record of the environment of the function definition (static link). A (code pointer, static link) pair can be passed around arbitrarily as the value of various function variables; it will be used to set up an activation record when evaluation of a function variable having it as value is requested. The static link of an activation record, which derives from the static link of a function value, is used to access global variables according to the block structure (i.e., function definition structure) of the program. (Incidentally, the static link of an activation record should not be confused with the “dynamic link” of an activation record. Dynamic links chain together the activation records according to function calls rather than function environments.)

Use of the contour model and its associated activation record retention allows functions values to be used freely as inputs and outputs of other functions and we truly have a function data type. Indeed one can specify additional operations besides evaluation, e.g., composition, within the \texttt{FUNCTION} space (see Reference 6 and Figures 9 and 10). (It should be emphasized that at any point in time only those activation records “accessible” (potentially still needed) are truly retained.)

APPLICATIONS

We present in this section several examples of application of the function data type. In most cases the program pieces have been abstracted from existing programs. They are categorized according to the characteristic application we wish to illustrate.

Isolation of Global Variables: The program piece given in Figure 2 illustrates the basic use of activation record retention. The variable \(\text{trace}\) is assigned to the output of the evaluation of a constant function. Here that output is a function. When \(\text{trace}\) is called, that function will be evaluated in an environment which includes the variable \(n\). Thus \(n\) is global to the \(\text{trace}\) function but hidden from all parts of the program which call \(\text{trace}\). (This application is reminiscent of the own-variable concept of Algol 60 in that only the \(\text{trace}\) routine itself and not its calls need be aware of the existence of \(n\).)
A similar application but with more levels appears in Figure 3. Here a function is assigned as value of SPACE.FORMER upon evaluation of a constant function. The subsequent evaluation of SPACE.FORMER then produces a space which has two hidden levels of environment to work under. Computation involved in forming the outer environment need only be repeated when the original constant function is reevaluated, while the inner environment will be recomputed, making use of the outer environment, at each call to SPACE.FORMER.

The essential characteristics of the function data type in these examples are retention and the fact that a function can change by virtue of a change in its environment. Following examples illustrate the effect of changing the value of a function variable to a new constant value.

**Stream Functions—Generation of Sequences:** Burge\(^6\) discusses an abstract scheme, called "streaming," for generating successive elements of an arbitrary sequence. The idea is that a function call returns one element of the sequence and a new "stream function" for generation of the next element and the next stream function. Successive calls to the sequence of stream functions successively produces the desired sequence. (This desired generation effect was accomplished by special "generation procedures" in an earlier version of Madcap.\(^4\)) The program of Figure 4 is a concrete realization of streaming using the function data type concept. The stream functions are successive values of the function variable new.composition. Each call of new.composition in the while statement produces the next composition (a composition of \(N\) into \(M\) parts is a vector \((c_1, c_2, \ldots, c_M)\) such that \(\sum c_i = N\)) as the value of the global variable \(c\) and changes the value of the variable new.composition. Each function evaluation also returns a boolean value to control the iteration. (Recall that the value of the last evaluated expression is returned from a function call.) The value true is returned while the generation is active and false is returned when the generation is complete.

A somewhat more involved example of streaming appears in Figure 6. This is a program piece, taken from the Madcap compiler itself, which forms the sequence of characters representing an identifier; the syntax diagram for identifier appears in Figure 5.

In Figure 6 the stream functions are successive values of the function variable stream. As before, these functions (1) return the sequence element as the value of a global variable, here of \(c\) (\(c\) is known to be either an upper or lower case letter upon entry to the initial value of stream), (2) assure that the value of the stream function is correct for the next generation, and (3) return true or false according as the generation remains active or is complete. In this example, the successive values of \(c\) are accumulated in a sequence using the structure former notation of Madcap.\(^5\)
When complete this sequence is assigned as the value of Identifier.

Coroutines: The function data type allows a high level realization of the idea of coroutines2.10 routines of equal stature that call each other and continue calculation at their previous point of departure when called themselves. A schematic illustration of this construction using Madcap notation appears in Figure 7.

In this scheme, two coroutines are implemented as the function variables f and g. A current evaluation of f changes f to a new value just prior to calling g and vice versa. As shown by the actual examples of this section, changing the value of a function may involve either changing the environment of a constant function value or assigning a new constant value to the function variable.

Backtrack Programming: The scanning of hierarchical structures, either actual or logical, is often called "backtrack programming." Backtrack programs usually take the form of a sequence of nested iterations, the nesting accom-
Implementation and Application of Function Data Type

plished by recursive procedure calls. Using function variables, a different organization is possible that allows the handling of the terminal nodes to be extracted from the depths of the recursion. An example using this organization appears in Figure 8. (While perhaps not truly backtracking, the hierarchical search structure of this simple problem is characteristic of all backtrack programs.) This program collects the terminal nodes of a tree into a sequence called A. For instance, if the structure 

```
(1, (2, 3, (4), 5), 6, (7, 8), 9, 10)
```

were used as input to g, then A would equal 

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

upon completion of the program. Each call to g produces a new environment for the successive activations of a deeper level of the tree, each activation being associated with one node of the tree. The parameters h save the function values to be used when backtracking.

```
f = «true»
x = terminal.type
g = «
s: {terminal.type: ? items}
h: f
l - 1
«
  i + i+1
  if i < #s:
    a + s_i
    if terminal a:
      x + a
      true
    else:
      f + g(a,f)
  »
else:
  f + h
«
```

```
f = g(S, «false»)
A = {x: while f}
```

Figure 8—Function variables in backtracking

The important feature of this example is that one terminal node (here, one value of x) is handled (or produced) by each call to the stream function (here, each evaluation of f during the formation of A).

Function spaces: The function and abstract data type concepts combine to permit the construction of spaces in which the elements are functions. A number of useful operations besides evaluation—composition, integration, differentiation—may be defined within such function spaces and referenced using natural notation—f g, f f, a g. In Figure 9 we illustrate partial function construction using the juxtaposition operator.

In this example, f is an element of a space of real valued functions of two real variables. The juxtaposition operator is used to specify the partial function formation as illustrated in the assignment of h, a real valued function of one real variable. The formation operation itself (see the function labeled ↓ jux) is a function of three formal parameters F, a, b; F has the same form as f, while a and b are real numbers. As shown, the value returned (last expression evaluated) is a function of one variable. The importance of retention is seen in this example, since when h itself is evaluated, it must do so within the environment of this formation in order to have access to the proper values of a or b.

Another potentially more important application of function spaces is suggested by the routine appearing in Figure 10. This is an abstract one-dimensional root finder routine. Its sole input is an arbitrary member, f, of a function space called FCT. All of the information needed to carry out the algorithm, left and right endpoints, error tolerance, etc., is extracted using operators (or function calls) of this space. The evaluation of f is specified using the natural juxtaposition operator, and the subtraction, multiplication, etc., operations of the argument space are also indicated with natural notation. Nevertheless, the precise form of the input function or type of its arguments are unspecified within this routine. That information is contained in the definition of the function space FCT and its environment. In a sense this algorithm is a higher level algorithm (a well-defined search) which is built upon lower level algorithms.
The function data type concept appears in programming languages as early as Lisp and is one of the advanced characteristics of Algol 68. It also appears unimplemented in the work of Burge and, in a somewhat different form using sets, in SETL. Nevertheless, partly because of notational questions, this concept has not received the attention befitting such an important unifying idea.

We believe that the use of activation record retention, a simple bracketed form for function constants, and the juxtaposition operator makes our scheme particularly natural and easy to use. Furthermore, our scheme benefits from and enhances other notational features of Madcap: (1) The type propagation scheme, which obviates explicit declarations, makes it more convenient to write algorithms that are completely independent of their data (Figure 10). (2) The structure former notation along with streaming provides a natural mechanism for constructing arbitrary sets, sequences and other structures (Figures 6, 8). (3) The value returned by a function being the last evaluated expression of the function body permits a concise yet readable form for deferred evaluation of expressions (Figure 8).

The existing Madcap compiler which implements the function data type described here does not possess a complete type-checking system. This deficiency is due both to expediency and to a lack of understanding with regard to type-checking for function values. However, we now feel reasonably confident that the various values assigned to a function variable must at least have the same number of input parameters and output parameters. Also, corresponding parameters should, in some sense, have the same type. The question of global variables is less clear. We have determined that for certain optimizations to be applicable, the compiler must be able to determine all global variables referenced by any value of each function variable.

There is much work to be done with regard to compiler construction and optimization for languages with a function data type. Nevertheless, we consider the function data type to be a significant step in the quest for a very general, concise, and readable scientific programming language.

REFERENCES

11. Rechard, O., (private communication).