Adverbial Programming

Michael J. A. Berry
(mjab@think.com)
Thinking Machines Corporation
245 First Street
Cambridge, MA 02142

Introduction

Sigplan Notices Volume 24, Number 11 contained an article by Joseph De Kerf entitled "A Note on the Power Operators ('Loops are Harmful')" which examines the way one particular APL conjunction may be used in place of a particular class of loops. Robert Mandl's response in *Sigplan Notices* Volume 25, Number 4 shows that, perhaps due to De Kerf's focus on a single example, a most important point was apparently lost: Adverbs and conjunctions which control the manner in which functions are applied to data provide a *replacement* for loops and similar control structures. In this note I briefly describe the adverbial approach to programming and contrast it with the approach of constructing ever more baroque serial control flow constructs such as the proposed `POWERLOOP`. The adverbial paradigm is especially relevant to the effort to develop good data parallel programming languages for the increasingly important class of massively parallel processors.

`POWERLOOP` *and its predecessors*

The `POWERLOOP` proposed by Mandl and illustrated in [4] in the contexts of Modula-2, Pascal, and C is not a new idea. Under the name `FORALL` it long formed part of the draft proposed ANSI and ISO standards for Fortran. Although it has now been dropped from the draft proposed standard, it has been implemented in CM Fortran™. This facility which performs a cartesian product of the values taken on by its index variables has proved quite useful. On the Connection Machine® massively parallel computer, the loop indices are used generate a particular pattern of interprocessor communication to spread arrays in the body of the `FORALL` to a rank sufficient to allow performing the seemingly scalar expressions in the `FORALL` body in parallel.

Powerful though it is, the `FORALL` or `POWERLOOP` only addresses one of a very large set of potentially interesting patterns. Trying to add new statement types for each of these interesting patterns would lead to unacceptable clutter. Fortunately, a superior approach exists.

Conjunctions and Adverbs

In the dialect of APL defined by Iverson's "A Dictionary of APL" [3], the terms "conjuction" and "adverb" are used to describe what have traditionally been called "operators" in APL, or "functional forms" in FP[1]. I use Iverson's terminology here both to avoid confusion with the term operator as used in other programming languages and to emphasize that creating new functions by modifying existing ones is at the heart of this approach to programming.

An adverb takes a verb as its argument and produces a new one which may be applied immediately or passed as the argument to another adverb or conjunction. Conjunctions are similar to adverbs, but they take two arguments, one of which may be a noun. Examples of adverbs include reduction, scan, and merge. Examples of conjunctions include various types of functional composition, inner and outer products, and a partitioning conjunction.

In adverbial programming, one defines functions which will do the right thing when applied to an entire collection of data rather than iterating through each scalar component of the data using control constructs to decide what to do at each point. The adverbial approach is particularly appropriate to the distributed memory SIMD environment. Adverbs may be thought of as establishing patterns of communication between processors. The functional arguments to the adverbs determine what is to be done to the data when it gets where it is going.

There are several important differences between the adverbial approach, and the approach of adding control constructs. Each control construct is, in effect, a miniature programming language of its own with a syntax different from that of every other control construct and from the rest of the language. Control constructs may be nested, but they cannot be modified the way the result of one adverbial expression may be modified by another. Control constructs generally require that knowledge of the data to be processed (the rank and shape of arrays, for example) be embedded in the code leading to maintainability headaches when the program is used in a new setting. Furthermore, control constructs such as the proposed `POWERLOOP` disguise parallelism by focusing on one element at a time. A smart compiler may be able to determine that what appear to be serial loops may, in fact, be executed in parallel, but there is a greater semantic distance between the parallel algorithm and its representation as a program.

The Adverbial Approach to Mandl's Telephone Number Example

The problem used to illustrate the usefulness of the POWERLOOP construct in [4] is to take a telephone number and generate all possible patterns of letters which represent that number on a standard U.S. telephone keypad.

In Mandl's example program, the data encoding the correspondence of letters to keys is embedded in the program. The length of a telephone number and the number of letters on a key are hard-coded in the loop parameters and in various other places in the program including the comments.

The dictionary APL function to accomplish the same task makes use of the outer product and rank conjunctions and the reduction adverb each of which is described below. All three are useful in a wide variety of situations.

The heart of the problem is to form strings by joining together 1 letter from each key. Without modification, the APL primitive verb for catenation does not quite do the right thing.

```
      'cat','hat'
cathat
```

The problem is that by default the catenation verb acts on the entirety of its arguments. We can use the rank conjunction to derive a related function having rank 0, meaning that it partitions its arguments into scalar cells. It is worth noting that the partitioning performed by the rank conjunction is analogous to partitioning an array into in-processor and across processor axes on a distributed memory SIMD computer.

```
      'cat' ,⍤0 'hat'
ch
aa
tt
```

Alternatively, we may derive a function which partitions its arguments into cells having rank one less than that of the arguments by using the rank conjunction with an argument of 1. For this example, the two functions return the same result, but the latter will prove to be more useful in treating the problem at hand.

```
      'cat' ,⍤¯1 'hat'
ch
aa
tt
```

Now, instead of pairing each cell in the left argument with the corresponding cell of the right argument, we would like each cell of the to be paired with every other. This is accomplished using an outer product.

```
      'cat' ∘.(,⍤¯1) 'hat'
ch
ca
ct
ah
aa
at
th
ta
tt
```

All that remains is to employ the reduction adverb ⌿ to derive a function which will perform this outer product succesively between adjacent cells of an array representing the letters on the telephone keys. Because the catenation at the heart of the algorithm is performed with rank ¯1, each (rank 0) single letter from the (rank 1) cell representing the Nth digit is joined to each of the (rank N-1) cells generated from the previous digits.

The final function definition is:

```
tele_words: ∘.(,⍤¯1)⌿⍵
```

Unlike the programs in [4], this one will work on telephone numbers of any length. All assumptions about what symbols are associated with which keys are in the data so the function need not be edited to for use with foreign phones. More importantly, the new function *tele_words* may itself be modified by adverbs so as, for example, to apply it to the first 3 digits and second 4 digits of a 7-digit number separately.

The following examples (run on a SUN 3 under SAX, an APL interpreter from I.P. Sharp Associates) use data from a pre-defined array called *tel* which represents the keyset used by Mandl. The 3 digit number 328 yields EAT and FAT as possible mnemonics. The second example augments this list with the numbers themselves in order to

be able to generate mnemonics of the form 1-800-2FRANCE used by Air France, or 1-800-4CIRRUS used by one of the ATM networks.

```
      tel
000
111
ABC
DEF
GHI
JKL
MNO
PRS
TUV
WXY
     telewords 3 2 8{tel                    telewords 3 2 8{tel,⍳10
DAT                                    D A T
DAU                                    D A U
DAV                                    D A V
                                       D A 8
DBT
DBU                                    D B T
DBV                                    D B U
                                       D B V
DCT                                    D B 8
DCU
DCV                                    D C T
                                       D C U
                                       D C V
EAT                                    D C 8
EAU
EAV                                    D 2 T
                                       D 2 U
EBT                                    D 2 V
EBU                                    D 2 8
EBV
ECT                                    E A T
ECU                                    E A U
ECV                                    E A V
                                       E A 8
FAT                                    E B T
FAU                                    E B U
FAV                                    E B V
                                       E B 8
FBT
FBU                                    E C T
FBV                                    E C U
                                       E C V
FCT                                    E C 8
FCU
FCV                                    E 2 T
                                         .
                                         .
                                         .
```


[1] Iverson, K.E. "A Dictionary of APL" *APL Quote-Quad* Vol. 18, Number 1, September 1987.

[2] Backus, John "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", *Communications of the ACM* Vol. 21, Number 8. August 1978.

[3] De Kerf, J.L.F "A Note on the Power Operator ('Loops are Harmful')", *Sigplan Notices* Vol. 24, Number 11. November 1989

[4] Mandl, R. "On 'POWERLOOP' Constructs in Programming", *Sigplan Notices* Vol. 25, Number 11.