Functional Programming

Gunnar Gotshalks
2003 September 27

Contents

Introduction

In the 1977 Turing Lecture John Backus described functional programming. As he put it "The problem with 'current languages' is that they are a word-at-a-time". Notable exceptions of the time were Lisp and APL. What we want to do is to think and work in units meaningful to the application and not to the underlying hardware. The analogy with word processing is not to work with words but to work with paragraphs, sections, chapters and even books at a time as appropriate.

Word-at-a-time translates to byte-at-a-time in modern jargon. In the early days of computing the "word" was the fundamental unit of storage. A word typically held an integer and depending upon the type of computer could be from 2 to 8 bytes in length.

To enable working with larger data units it is necessary to abstract the control flow patterns and give them names to easily reuse the control flow pattern over and over again. For example in most languages when working with an array we write loops to explicitly process each element of the array. If we abstract the loop control flow we can think of processing the entire array as a single operation.

Example for Inner Product

A functional definition of the inner product of two vectors is given by the following functional form [1].

Given vector A = $<a_1, \ldots, a_n>$ and vector B = $<b_1, \ldots, b_n>$. the inner product of A and B is $(a_1 * b_1 + a_2 * b_2 + \ldots + a_n * b_n)$.

---

innerProduct ::= (/+) ∘ (α x) ∘ trans

Functional programs have no explicit loops (but can have recursive definitions), have no sequencing at a low level, have no local variables. Functional programming consists of writing functions that have functions as as input and frequently as output. That is writing functions that themselves create new functions.

One applies a functional, in Backus' notation, to a single argument consisting of a list of the actual arguments – on the surface every functional has only one argument. For example assuming we have two vectors $<a_1, \ldots, a_n>$ and $<b_1, \ldots, b_n>$, then the inner product is applied using the following form.

innerProduct : $< <a_1, \ldots, a_n> <b_1, \ldots, b_n> >$

Explanation of example

trans

Transposes the 2xn matrix to an nx2 matrix by interchanging rows and columns. The result being $<<a_1, b_1> <a_2, b_2> \ldots <a_n, b_n>>$. Notice that we still have a single list as an argument to next part of the functional.

(α x) – read as apply times

Applies the multiplication operator "x" to each pair of elements in its argument list producing $< a_1 * b_1, a_2 * b_2, \ldots, a_n * b_n >$

(/+) – read as reduce using +

Puts the plus operator between each pair of elements in the argument list and does the summation which gives us the inner product $(a_1 * b_1 + a_2 * b_2 + \ldots + a_n * b_n)$.

"o" – the function composition operator

"f o g : x" is equivalent to f(g(x)). It tells us to apply the functions in the functional form from right to left with the output of each function being passed as input to the next function – like a pipeline.

At the end of this document is a section that describes various Lisp solutions to finding the inner product that are based on the above functional definition.

Basis Set for Functionals

Type of object	**Lisp notation**	**Backus notation**
List	(a b c d)	<a, b, c, d>
Constant	literal	literal
Builtin functions	**+, *, etc.**	+, x, etc.
Selector functions	cdr, first, second, ...	**tail, 1, 2, ...**
Constructor functions	cons	[f1 , f2 , ... , fn] f1..fn each operate on the input to produce a list of results of length n
Function application	(f x) (apply f (x)) (funcall f x)	f : x
Mapping functions	(map f ...) (mapcar f ...) (maplist f ...)	(α f)
Reduction	(reduce f a)	/f
Function composition	(comp f g)	f $\circ$ g
Binding(currying)	(bu f k)	(bu f k)
Choice function	cond	predicate $\rightarrow$ f-true ; f-false
Constant function	literal	$\overline{\text{literal}}$

Example Function Definitions

Adding two numbers

```
add ::= +      Effect is to rename +
add : < 2, 3>   ==>  5
```

Adding a list of numbers

```
addlist ::= /+
addlist : < 2, 3, 4, 5, 6>   ==>  20
```

Absolute value

```
abs ::=  (bu > 0) → - ; id
abs : -1   ==>  1          abs : 1   ==>  1
```

The expression (bu > 0) curries the function > with the first parameter fixed to the number 0, thereby creating a predicate that evaluates 0 > argument. If the predicate is true then the function - , unary minus, is applied to the argument. If the predicate is false then the function id, identity, is applied to the argument.

Factorial

An example showing a recursive definition – note that function names can be strings of symbols.

```
! ::= (bu = 0) → 1 ; x o [id , ! o (bu (rev -) 1)]
```

Function definitions need to be understood as they apply to arguments. Consider the expression ! : 2 that evaluates to 2!. The expression (bu = 0) creates the predicate to compare the argument to 0. 2 is not equal to 0, so the function

```
x o [id , ! o (bu (rev -) 1)]
```

is applied to 2 giving

```
x o [id , ! o (bu (rev -) 1)] : 2
x o [id : 2 , ! o (bu (rev -) 1) : 2]
```

id just returns the argument while the expression (bu (rev -) 1) creates the subtract one function, so we have the following where ! is applied to 1 (= 2 - 1)

```
x o [2 , ! : 1 ]
```

! is applied to 1, and since 1 not equal to 0, we have

```
x o [2, x o [id , ! o (bu (rev -) 1) ] : 1]
x o [2, x o [id : 1 , ! o (bu (rev -) 1) : 1]]
x o [2, x o [1 , !  :  0]]
```

! is applied to 0 and since 0=0 the result is the application of the constant function

1 to the argument 0, which returns 1 (the constant value the function returns irregardless of the argument value)

```
x o [2, x o [1 , 1]]
```

The inner [] becomes the list <1, 1> to which the function x is applied

```
x o [2, x : <1, 1>]
```

We now have

```
x o [2, 1]
x : <2, 1>
2
```

The definition of ! is seen as being clumsy because of the construction of the low level equal to 0 and subtract by 1 functions. In a functional programming environment a set of such functions would be available in a library (for example the 1+ and 1- functions in Lisp). The following shows the definition of factorial as a sequence of auxiliary definitions of which only the last definition would be necessary assuming the first two definitions are available.

```
=0 ::= (bu = 0)
1- ::= (bu (rev -) 1)
!  ::=  =0 → 1 ; x o [id , ! o 1-]
```

Using the list constructor

One of the techniques in writing functional programs is to use the list constructor in building up larger expressions from smaller expressions. For example consider a function to evaluate the following function.

```
fn1 : <a, b, c, d> ≡  (a + b) ÷ (c - d)
```

You can build the definition from the smaller expressions to the larger expressions.

```
(a + b) ::= + o [ 1 , 2 ]  and  (c - d) ::= - o [ 3 , 4 ]
```

And then combine into larger expressions.

```
fn1 ::= ÷o [ + o [ 1 , 2 ] , - o [ 3 , 4 ] ]
```

Lisp Builtin Functionals

Function application

(**apply** function (argList))
direct analogy with Backus form where the function has one argument list.

(**funcall** function arg_1 arg_2 ... arg_n)
number of arguments is the number required by the function.

Mapping and reduction

The number of arguments supplied to the mapping functions is one (function to map) plus the number of arguments required by the mapped function.

(**mapcar** function arg_1 arg_2 ... arg_n)
Apply the function of n-arguments to the first element of each list, then the second element, etc. Continue until one of the arguments is the empty list. Combine all the results into one list.

```
(mapcar '1+ '(1 2 3))  ==>  (2 3 4)
(mapcar 'cons '(a b c) '(1 2 3))  ==>  ((a.1) (b.2) (c.3))
```

(**maplist** function arg_1 arg_2 ... arg_n)
Apply the function of n-arguments to all the arguments, then remove the first element of each argument, apply the function to the shortened arguments, remove the first element, apply the function to the shortened arguments, etc. Continue until an argument is the empty list. All results are combined into one list.

```
(maplist f '(a b c) '(1 2 3))
    ==>  ((f '(a b c) '(1 2 3)) (f '(b c) '(2 3)) (f '(c) '(3)))

(maplist 'cons '(a b c) '(1 2 3)
    ==>  ( ((a b c) 1 2 3)   ((b c) 2 3)   ((c) 3) )
Contrast with (mapcar 'cons '(a b c) '(1 2 3)) shown above.
```

(**reduce** binaryFunction argList)
Apply the binaryFunction to the first two arguments in the argument list. The function is then applied to the result and the next item in the list until the list is exhausted. The items in the list must of the correct type for the function. Test what happens if the empty list or a one item list is input.

```
(reduce '+ '(1 2 3 4))  ==>  10
(reduce '* '(1 2 3 4))  ==>  24
(reduce 'append '((a) (b) (c) (d)))  ==>  (a b c d)
```

A Lisp Functional Library

The following are basic functionals that are not built into Lisp but are convenient to have.

(bu binaryFunction argument)

Bind the first parameter of a binary function to a fixed value, thereby creating a function of one parameter. The effect is to convert a binary function to a unary function. This is also called currying from the mathematician Curry who developed the idea. Currying is done in everyday language. For example, when we say "Susan is tall" we are using a unary function "is tall" which is derived from the binary function "a is taller than b" by binding the variable b a constant.

(bu '+ 3) creates a unary function to add 3 from the binary function "+".

```
(mapcar(bu '+ 3) '(1 2 3))  ==>  (4 5 6)
```

(bu 'cons 'x) creates a unary function to cons x with its argument.

```
(mapcar(bu 'cons 'x) 1)  ==>  (x.1)
```

To define a function 3+ using bu do the following.

```
(defun 3+ (x) (funcall (bu '+ 3) x))
Then   (mapcar '3+ '(1 2 3))  ==>  (4 5 6)
```

To define a function consX using bu do the following.

```
(defun consX (x) (funcall (bu 'cons 'x) x))
Then   (consX 1)  ==>  (x.1)
```

Definition

```
(defun bu (f x)
 (function (lambda (y) (funcall f x y))))
```

Why do we need the function bu? Why not simply write a function such as 3+ as in the following? It is shorter and easier to write.

```
(defun 3+ (x) (+ 3 x))
```

This works perfectly well. For example, we can use the function to add 3 to every member of a list of numbers with the following.

```
(mapcar '3+ '(1 2 3 4))  ==>  (4 5 6 7)
```

If the function 3+ happens to be around then all is well but if we define 3+ just for the above mapcar and have no other use for it, it is a waste.

Instead we can use a lambda function and create a one shot unnamed function within the mapcar construct as follows, where `#'` is shorthand for `(function ... )`.

```
(mapcar #'(lambda(x)(+ 3 x)) '(1 2 3 4)) ==> (4 5 6 7)
```

The use of a lambda function is seen as being clumsy. Using bu we have the following shorter less complex program text. Can also do this when the function definition is not available to us.

```
(mapcar (bu '+ 3) '(1 2 3 4)) ==> (4 5 6 7)
```

(rev binaryFunction)

The functional rev creates a binary function similar to the input function but with the parameters in the reverse the order. In the examples for the function bu we defined consX such that `(consX '1) = (x.1)`. Suppose you want `(1.x)`. You could define yet another function or you could use rev as follows.

```
(funcall (rev consX) 1)
```

To define a new function rconsX you can do the following.

```
(defun rconsX (x) (funcall (bu (rev 'cons) 'x) x))
```

And use it as follows.

```
(rconsX 1)  ==>  (1.x)
```

Definition

```
(defun rev (f)
 (function (lambda (x y) (funcall f y x))))
```

(comp function1 function2)

comp creates a unary function as the composition of a pair of unary functions. For example an expensive "donothing" to add one and then subtract one.

```
(defun donothing (x) (funcall (comp '1- '1+) x))
```

For example use of comp see versions 3 and 4 of inner product described later in the document.

Definition

```
(defun comp (f g)
  (function (lambda (x) (funcall f ( funcall g x)))))
```

(compl f1 f2 ... fn)

compl creates a unary function from the composition of a sequence of unary functions.

The above donothing example can be constructed using compl as follows.

```
(defun donothing2 (x) (funcall (compl '1- '1+) x))
```

Now do a plus 7 using 3+ and 1+.

```
(defun 7+ (x) (funcall (compl '3+ '3+ '1+) x))
```

Definition

```
(defun compl (&rest funlist)
  (function (lambda (x) (compbody x funlist))))

(defun compbody (x &rest funlist)
  (cond ((null (car funlist)) x)
        (t (funcall (caar funlist) (compbody x (cdar funlist))))))
```

Utility list reordering functions

We describe three functions that reorder list elements to get them into the right order or shape expected by specific functionals. For example the function rev presented earlier is used specifically for functionals that require the order of two arguments to be reversed.

(trans matrix)

Transpose (swap rows and columns) of a two-dimensional matrix.

```
((a b c) (d e f)) <=> ((a d) (b e) (c f))
```

In matrix form it looks as follows.

```
a b c   <=>    a d
d e f          b e
               c f
```

Definition

```
(defun trans (matrix) (apply 'mapcar 'list matrix))
```

(distl anItem theList)
(distlr anItem theList)
(distl1 anItem theList)

Distribute left distributes the first parameter, anItem, to the left of each of the items in the second parameter, theList.

```
(distl 'a '(1 2 3))  ==>  ((a 1) (a 2) (a 3))
```

Definitions

Distribute left recursive version.

```
(defun distlr (x b)
 (cond ((null b) nil)
       (t (cons (list x (car b)) (distl x (cdr b))))))
```

Functional two parameter version.

```
(defun distl (x b)
  (mapcar (function (lambda (y) (list x y))) b))
```

Functional one parameter version in the style of Backus.

```
(defun distl1 (x-b)
  (mapcar (function (lambda (y) (list (first x-b) y)))
                                      (second x-b)))
```

(distr anItem theList)
(distrr anItem theList)
(distr1 anItem theList)

Distribute right distributes the first parameter, anItem, to the right of each of the items in the second parameter, theList.

```
(distr 'a '(1 2 3))  ==>  ((1 a) (2 a) (3 a))
```

Definitions

Distribute right recursive version.

```
(defun distrr (x b)
 (cond ((null b) nil)
       (t (cons (list (car b) x) (distrr x (cdr b))))))
```

Functional two parameter version.

```
(defun distr (x b)
 (mapcar (function (lambda (y) (list y x))) b))
```

Functional one parameter version in the style of Backus.

```
(defun distr1 (x-b)
  (mapcar (function (lambda (y) (list y (first x-b))))
                                        (second x-b)))
```

Lisp Implementation of Inner Product

A variety of solutions to calculating the inner product of a pair of vectors to show different styles of Lisp programs from fully recursive to using functionals. The inner product of two vectors is the sum of the products of the corresponding elements.

innerProduct : $< <a_1, \ldots, a_n> <b_1, \ldots, b_n> >$ ==> $(a_1 * b_1 + a_2 * b_2 + \ldots + a_n * b_n)$

Backus' functional definition [1] is the following.

innerProduct ::= $(/+) \circ (\alpha\ x) \circ$ trans

Version 0

The following is a recursive version without using functionals and having two arguments. Recursion proceeds down both arguments simultaneously.

```
(defun ip0a (a b)
  (cond ((null a) 0)
        (t (+ (* (car a) (car b))
              (ip0a (cdr a) (cdr b))))
))
```

The following is the one argument version in the style of Backus functionals. The solution is more complex because the pair must be torn apart and rebuilt on each recursive call.

```
(defun ip0b (a-b-pair)
  (cond ((null (car a-b-pair)) 0)
        (t (+ (* (caar a-b-pair) (caadr a-b-pair))
              (ip0b (list (cdar a-b-pair)
                          (cdadr a-b-pair)))))
))
```

Version 1

The following is a definition with two arguments. Notice the parallel with Backus' definition. In this case function composition is done by explicit nesting of the functions. Due to the special nature of transpose, mapcar does both the transpose and multiplicative combining of the two vectors.

```
(defun ip1 (a b) (reduce '+ (mapcar '* a b)))
```

Version 2

The following is a definition using a single argument that is a list containing both vectors as sublists. The change from the previous definition is to extract the two vectors from the single list.

```
(defun ip2 (a-b-pair)
  (reduce '+ (mapcar '* (first a-b-pair) (second a-b-pair))))
```

Version 3

The following definition explicitly uses function composition. The argument is again a single list with the vectors as sublists.

```
(defun ip3 (a-b-pair)
  (funcall (comp '/+ (comp 'A* 'trans)) a-b-pair))
```

Where we define the functions "/+" (reduce using +) and A* (apply * to all).

```
(defun A* (a-b-pair) (mapcar '*pair a-b-pair))
(defun /+ (a-b-pair) (reduce '+pair a-b-pair))
```

We further define "*pair" and "+pair" to do multiplication and addition on lists containing a pair of elements from a and b.

```
(defun *pair (a-b-pair) (* (first a-b-pair) (second a-b-pair)))
(defun +pair (a-b-pair) (+ (first a-b-pair) (second a-b-pair)))
```

Version 4

A transliteration of Backus' definition into Lisp using, comp, a composition of a pair of functions. Since composition occurs twice in Backus' definition, comp is nested. Note that function composition associates from right to left. "trans" is assumed to have one argument being a pair of vectors.

```
(defun ip4 (a-b-pair)
  (funcall (comp (bu 'reduce '+)          ; (/+) for Backus
```

```
                (comp (bu 'mapcar '*pair) ; (alpha *) for Backus
                      'trans))            ; trans for Backus
                           a-b-pair))     ; apply to the pair of vectors
```

Why (bu 'mapcar '*pair)? Recall that the basic definition of mapcar is a function with two arguments — (mapcar functionToApply listsToApplyTo). (bu 'mapcar '*pair) creates an unnamed unary function which combines mapcar and *pair to create a customized function. Lets give it the alias mapcar-functionToApply, which can be invoked as follows.

```
(mapcar-functionToApply listsToApplyTo)
```

The unary function can then be composed with the unary function trans. Similarly the binary function reduce is turned into a unary function (/+) using (bu 'reduce '+).

To get this example to execute you need a transpose function that accepts one argument.

Version 5

A transliteration of Backus' definition into Lisp using, compl, a composition of a list of functions (/+), (alpha *) and trans, rather than nesting function composition.

```
(defun ip5 (a-b-pair)
  (funcall (compl (bu 'reduce '+)      ; (/+) for Backus
                  (bu 'mapcar '*pair)  ; (alpha *) for Backus
                   'trans)             ; trans for Backus
                        a-b-pair))     ; apply to the pair of vectors
```

The inner body (reproduced in the following) is a close match to Backus' notation.

```
(bu 'reduce '+)
(bu 'mapcar '*pair)
'trans
```

The outer shell (reproduced in the following) provides the Lisp environment for executing the functions.

```
(defun ip5 (a-b-pair)
  (funcall (compl ...
                  ...   )
                        a-b-pair))
```

Using this style one can easily transliterate Backus notation into executable Lisp programs.

All Pairs Functional Definition

One allPairs specification is the following..

```
allPairs : < <a, b, c> <1, 2, 3, 4> > ==>
      < <a,1> <a,2> <a,3> <a,4>
        <b,1> <b,2> <b,3> <b,4>
        <c,1> <c,2> <c,3> <c,4>
      >
```

How can you derive a functional expression that computes all pairs? Start out with the initial state as a "picture" and think about which functionals you know that may bring you to a picture closer to the final picture. Initially we have the following input consisting of a list of two lists.

```
input = < <a, b, c> <1, 2, 3, 4> >
```

The final state has the following picture; a list of all pairs from the initial two lists.

```
< <a,1> <a,2> <a,3> <a,4>
  <b,1> <b,2> <b,3> <b,4>
```

```
  <c,1> <c,2> <c,3> <c,4>
>
```

Of all the functionals described in the library, distribution seems to be the closest. Lets see what it does from the initial state. In the result elements from the first list is on the left so try distribute left.

distl : input ==> `< <<a,b,c> 1> <<a,b,c> 2> <<a,b,c> 3> <<a,b,c> 4> >`

This looks good because if we can distribute the numbers over each of the "letter lists" we have the result. The problem is distribute distributes the first argument over the list but, in the above, we want to distribute the second argument over the first argument. It looks like rev might be useful but we notice we have the distribute right available which seems to be simpler.

distr : input ==> `< <1 <a,b,c>> <2 <a,b,c>> <3 <a,b,c>> <4 <a,b,c>> >`

If we distribute right on each list in the above we are very close to the answer. But even looking at the partial result we see the first pairs have the "1" repeated in the first few tuples while the specification has the "a" repeated. It looks like we want to swap the input lists first, then do the distribute right. Lets see.

distr ∘ [**2** , **1**] : input ==> `< <a <1 2 3 4>> <b <1 2 3 4>> <c <1 2 3 4>> >`

Now it looks like we need to distribute left over each sublist – we need <a,1> and not <1,a>. Operating over each sublist implies doing a "for all". We now have.

(α distl) ∘ distr ∘ [**2** , **1**] : input ==>

```
     < < <a,1> <a,2> <a,3> <a,4> >
       < <b,1> <b,2> <b,3> <b,4> >
       < <c,1> <c,2> <c,3> <c,4> >
>
```

It looks very close to the answer except that it is composed of three sublists. We need to combine all the sublists into one list. The reduce functional combines elements of a list into one. Which binary function do we need? Thinking about how reduce is defined we note we are appending each sublist to an every growing result list. Thus we have the following.

/append ∘ (α distl) ∘ distr ∘ [**2** , **1**] : input ==>

```
     < <a 1> <a 2> <a 3> <a 4)
       <b 1> <b 2> <b 3> <b 4>
       <c 1> <c 2> <c 3> <c 4> >
```

Finally we have the following functional definition to computing all pairs from two lists.

allPairs ::= /append ∘ (α distl) ∘ distr ∘ [**2** , **1**]

The above definition produces the order given in our example. Other orderings are possible as follows using other combinations swapping or not swapping the initial sublists and using left or right distribution at the second distribution step.

allPairs ::= /append ∘ (α distr) ∘ distr ∘ [**2** , **1**]
allPairs ::= /append ∘ (α distl) ∘ distr
allPairs ::= /append ∘ (α distr) ∘ distr

Matrix Multiplication Functional Definition

The following is a functional definition of matrix multiplication from [1] – where ip is inner product, distl is distribute left, distr is distribute right, and trans is transpose.

matmult ::= (α α ip) ∘ (α distl) ∘ distr ∘ [**1** , trans ∘ **2**]

The following is a functional based lisp definition from [2].

```
(defun matProd (a b) (mapcar (bu 'prodRow (trans b)) a))
(defun prodRow (bt r) (mapcar (bu 'ip r) bt))
```

References

1 John Backus, "*1977 Turing Award Lecture: Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs*", CACM August 1978, V21, No 8, pp 613-641.

2 Bruce MacLennan, "*Principles of Programming Languages*", Holt, Rhinehart, Winston, 1987, second edition, ppp 382-397.