Why Functional Programming Matters*

John Hughes, Institutionen för Datavetenskap,
Chalmers Tekniska Högskola,
41296 Göteborg,
SWEDEN. rjmh@cs.chalmers.se

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs. Conventional languages place conceptual limits on the way problems can be modularised. Functional languages push those limits back. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). Since modularity is the key to successful programming, functional languages are vitally important to the real world.

1 Introduction

This paper is an attempt to demonstrate to the "real world" that functional programming is vitally important, and also to help functional programmers exploit its advantages to the full by making it clear what those advantages are.

Functional programming is so called because a program consists entirely of functions. The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. These functions are much like ordinary mathematical functions, and in this paper will be defined by ordinary equations. Our notation follows Turner's language Miranda(TM) [Tur85], but should be readable with no prior knowledge of functional languages. (Miranda is a trademark of Research Software Ltd.)

The special characteristics and advantages of functional programming are often summed up more or less as follows. Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant - since no side-effect can change the value of an expression, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa - that is, programs are "referentially transparent". This freedom helps make functional programs more tractable mathematically than their conventional counterparts.

Such a catalogue of "advantages" is all very well, but one must not be surprised if outsiders don't take it too seriously. It says a lot about what functional programming is *not* (it has no

*This paper dates from 1984, and circulated as a Chalmers memo for many years. Slightly revised versions appeared in 1989 and 1990 as [Hug90] and [Hug89]. This version is based on the original Chalmers memo `nroff` source, lightly edited for LaTeX and to bring it closer to the published versions, and with one or two errors corrected. Please excuse the slightly old-fashioned type-setting, and the fact that the examples are not in Haskell!.

assignment, no side effects, no flow of control) but not much about what it is. The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous. To those more interested in material benefits, these "advantages" are not very convincing.

Functional programmers argue that there *are* great material benefits - that a functional programmer is an order of magnitude more productive than his conventional counterpart, because functional programs are an order of magnitude shorter. Yet why should this be? The only faintly plausible reason one can suggest on the basis of these "advantages" is that conventional programs consist of 90% assignment statements, and in functional programs these can be omitted! This is plainly ridiculous. If omitting assignment statements brought such enormous benefits then FORTRAN programmers would have been doing it for twenty years. It is a logical impossibility to make a language more powerful by omitting features, no matter how bad they may be.

Even a functional programmer should be dissatisfied with these so-called advantages, because they give him no help in exploiting the power of functional languages. One cannot write a program which is particularly lacking in assignment statements, or particularly referentially transparent. There is no yardstick of program quality here, and therefore no ideal to aim at.

Clearly this characterisation of functional programming is inadequate. We must find something to put in its place - something which not only explains the power of functional programming, but also gives a clear indication of what the functional programmer should strive towards.

2 An Analogy with Structured Programming

It is helpful to draw an analogy between functional and structured programming. In the past, the characteristics and advantages of structured programming have been summed up more or less as follows. Structured programs contain no **goto** statements. Blocks in a structured program do not have multiple entries or exits. Structured programs are more tractable mathematically than their unstructured counterparts. These "advantages" of structured programming are very similar in spirit to the "advantages" of functional programming we discussed earlier. They are essentially negative statements, and have led to much fruitless argument about "essential **gotos**" and so on.

With the benefit of hindsight, it is clear that these properties of structured programs, although helpful, do not go to the heart of the matter. The most important difference between structured and unstructured programs is that structured programs are designed in a modular way. Modular design brings with it great productivity improvements. First of all, small modules can be coded quickly and easily. Secondly, general purpose modules can be re-used, leading to faster development of subsequent programs. Thirdly, the modules of a program can be tested independently, helping to reduce the time spent debugging.

The absence of **gotos**, and so on, has very little to do with this. It helps with "programming in the small", whereas modular design helps with "programming in the large". Thus one can enjoy the benefits of structured programming in FORTRAN or assembly language, even if it is a little more work.

It is now generally accepted that modular design is the key to successful programming, and languages such as Modula-II [Wir82], Ada [oD80] and Standard ML [MTH90] include features specifically designed to help improve modularity. However, there is a very important point that is often missed. When writing a modular program to solve a problem, one first divides the problem into sub-problems, then solves the sub-problems and combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase ones ability to modularise a problem conceptually, one must provide new kinds of glue in the programming language. Complicated scope rules and provision for separate compilation only help with clerical details; they offer no new conceptual tools for decomposing problems.

One can appreciate the importance of glue by an analogy with carpentry. A chair can be made quite easily by making the parts - seat, legs, back etc. - and sticking them together in the right way. But this depends on an ability to make joints and wood glue. Lacking that ability, the only way to make a chair is to carve it in one piece out of a solid block of wood, a much harder task. This example demonstrates both the enormous power of modularisation and the importance of having the right glue.

Now let us return to functional programming. We shall argue in the remainder of this paper that functional languages provide two new, very important kinds of glue. We shall give many examples of programs that can be modularised in new ways, and thereby greatly simplified. This is the key to functional programming's power - it allows greatly improved modularisation. It is also the goal for which functional programmers must strive - smaller and simpler and more general modules, glued together with the new glues we shall describe.

3 Glueing Functions Together

The first of the two new kinds of glue enables simple functions to be glued together to make more complex ones. It can be illustrated with a simple list-processing problem - adding up the elements of a list. We define lists by

```
listof X ::= nil | cons X (listof X)
```

which means that a list of Xs (whatever X is) is either nil, representing a list with no elements, or it is a cons of an X and another list of Xs. A cons represents a list whose first element is the X and whose second and subsequent elements are the elements of the other list of Xs. X here may stand for any type - for example, if X is "integer" then the definition says that a list of integers is either empty or a cons of an integer and another list of integers. Following normal practice, we will write down lists simply by enclosing their elements in square brackets, rather than by writing conses and nils explicitly. This is simply a shorthand for notational convenience. For example,

```
[]              means   nil
[1]             means   cons 1 nil
[1,2,3]         means   cons 1 (cons 2 (cons 3 nil))
```

The elements of a list can be added up by a recursive function *sum*. Sum must be defined for two kinds of argument: an empty list (nil), and a cons. Since the sum of no numbers is zero, we define

```
sum nil = 0
```

and since the sum of a cons can be calculated by adding the first element of the list to the sum of the others, we can define

```
sum (cons num list) = num + sum list
```

Examining this definition, we see that only the boxed parts below are specific to computing a sum.

```
          +---+
sum nil = | 0 |
          +---+
                          +---+
sum (cons num list) = num | + | sum list
                          +---+
```

This means that the computation of a sum can be modularised by glueing together a general recursive pattern and the boxed parts. This recursive pattern is conventionally called *reduce* and so sum can be expressed as

```
sum = reduce add 0
```

where for convenience reduce is passed a two argument function *add* rather than an operator. Add is just defined by

```
add x y = x + y
```

The definition of reduce can be derived just by parameterising the definition of sum, giving

```
(reduce f x) nil = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

Here we have written brackets around (reduce f x) to make it clear that it replaces sum. Conventionally the brackets are omitted, and so ((reduce f x) l) is written as (reduce f x l). A function of 3 arguments such as reduce, applied to only 2 is taken to be a function of the one remaining argument, and in general, a function of n arguments applied to only $m(< n)$ is taken to be a function of the $n - m$ remaining ones. We will follow this convention in future.

Having modularised sum in this way, we can reap benefits by re-using the parts. The most interesting part is reduce, which can be used to write down a function for multiplying together the elements of a list with no further programming:

```
product = reduce multiply 1
```

It can also be used to test whether any of a list of booleans is true

```
anytrue = reduce or false
```

or whether they are all true

```
alltrue = reduce and true
```

One way to understand (reduce f a) is as a function that replaces all occurrences of cons in a list by f, and all occurrences of nil by a. Taking the list [1,2,3] as an example, since this means

```
cons 1 (cons 2 (cons 3 nil))
```

then (reduce add 0) converts it into

```
add 1 (add 2 (add 3 0))  =  6
```

and (reduce multiply 1) converts it into

```
multiply 1 (multiply 2 (multiply 3 1))  =  6
```

Now it is obvious that (reduce cons nil) just copies a list. Since one list can be appended to another by consing its elements onto the front, we find

```
append a b = reduce cons b a
```

As an example,

```
append [1,2] [3,4] = reduce cons [3,4] [1,2]
                   = (reduce cons [3,4]) (cons 1 (cons 2 nil))
                   = cons 1 (cons 2 [3,4]))
                        (replacing cons by cons and nil by [3,4])
                   = [1,2,3,4]
```

A function to double all the elements of a list could be written as

```
        doubleall = reduce doubleandcons nil
where   doubleandcons num list = cons (2*num) list
```

Doubleandcons can be modularised even further, first into

```
        doubleandcons = fandcons double
where   double n = 2*n
        fandcons f el list = cons (f el) list
```

and then by

```
        fandcons f = cons . f
```

where "." (function composition, a standard operator) is defined by

```
        (f . g) h = f (g h)
```

We can see that the new definition of fandcons is correct by applying it to some arguments:

```
        fandcons f el = (cons . f) el
                      = cons (f el)
so      fandcons f el list = cons (f el) list
```

The final version is

```
        doubleall = reduce (cons . double) nil
```

With one further modularisation we arrive at

```
        doubleall = map double
        map f = reduce (cons . f) nil
```

where map applies any function f to all the elements of a list. Map is another generally useful function.

We can even write down a function to add up all the elements of a matrix, represented as a list of lists. It is

```
        summatrix = sum . map sum
```

The map sum uses sum to add up all the rows, and then the left-most sum adds up the row totals to get the sum of the whole matrix.

These examples should be enough to convince the reader that a little modularisation can go a long way. By modularising a simple function (sum) as a combination of a "higher order function" and some simple arguments, we have arrived at a part (reduce) that can be used to write down many other functions on lists with no more programming effort. We do not need to stop with functions on lists. As another example, consider the datatype of ordered labelled trees, defined by

```
        treeof X ::= node X (listof (treeof X))
```

This definition says that a tree of Xs is a node, with a label which is an X, and a list of subtrees which are also trees of Xs. For example, the tree

```
        1 o
         / \
        /   \
       /     \
    2 o       o 3
              |
              |
              |
              o 4
```

would be represented by

```
node 1
    (cons (node 2 nil)
          (cons (node 3
                     (cons (node 4 nil) nil))
                nil))
```

Instead of considering an example and abstracting a higher order function from it, we will go straight to a function redtree analogous to reduce. Recall that reduce took two arguments, something to replace cons with, and something to replace nil with. Since trees are built using node, cons and nil, redtree must take three arguments - something to replace each of these with. Since trees and lists are of different types, we will have to define two functions, one operating on each type. Therefore we define

```
redtree f g a (node label subtrees) =
        f label (redtree' f g a subtrees)
redtree' f g a (cons subtree rest) =
        g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a nil = a
```

Many interesting functions can be defined by glueing redtree and other functions together. For example, all the labels in a tree of numbers can be added together using

```
sumtree = redtree add add 0
```

Taking the tree we wrote down earlier as an example, sumtree gives

```
add 1
    (add (add 2 0)
         (add (add 3
                   (add (add 4 0) 0))
              0))
= 10
```

A list of all the labels in a tree can be computed using

```
labels = redtree cons append nil
```

The same example gives

```
cons 1
    (append (cons 2 nil)
            (append (cons 3
                          (append (cons 4 nil) nil))
                    nil))
= [1,2,3,4]
```

Finally, one can define a function analogous to map which applies a function f to all the labels in a tree:

```
maptree f = redtree (node . f) cons nil
```

All this can be achieved because functional languages allow functions which are indivisible in conventional programming languages to be expressed as a combination of parts - a general higher order function and some particular specialising functions. Once defined, such higher order functions allow many operations to be programmed very easily. Whenever a new datatype is defined higher order functions should be written for processing it. This makes manipulating the datatype easy, and also localises knowledge about the details of its representation. The best analogy with conventional programming is with extensible languages - it is as though the programming language can be extended with new control structures whenever desired.

4 Glueing Programs Together

The other new kind of glue that functional languages provide enables whole programs to be glued together. Recall that a complete functional program is just a function from its input to its output. If f and g are such programs, then (g . f) is a program which, when applied to its input, computes

```
g (f input)
```

The program f computes its output which is used as the input to program g. This might be implemented conventionally by storing the output from f in a temporary file. The problem with this is that the temporary file might occupy so much memory that it is impractical to glue the programs together in this way. Functional languages provide a solution to this problem. The two programs f and g are run together in strict synchronisation. F is only started once g tries to read some input, and only runs for long enough to deliver the output g is trying to read. Then f is suspended and g is run until it tries to read another input. As an added bonus, if g terminates without reading all of f's output then f is aborted. F can even be a non-terminating program, producing an infinite amount of output, since it will be terminated forcibly as soon as g is finished. This allows termination conditions to be separated from loop bodies - a powerful modularisation.

Since this method of evaluation runs f as little as possible, it is called "lazy evaluation". It makes it practical to modularise a program as a generator which constructs a large number of possible answers, and a selector which chooses the appropriate one. While some other systems allow programs to be run together in this manner, only functional languages use lazy evaluation uniformly for every function call, allowing any part of a program to be modularised in this way. Lazy evaluation is perhaps the most powerful tool for modularisation in the functional programmer's repertoire.

4.1 Newton-Raphson Square Roots

We will illustrate the power of lazy evaluation by programming some numerical algorithms. First of all, consider the Newton-Raphson algorithm for finding square roots. This algorithm computes the square root of a number N by starting from an initial approximation a0 and computing better and better ones using the rule

```
a(n+1) = (a(n) + N/a(n)) / 2
```

If the approximations converge to some limit a, then

```
        a = (a + N/a) / 2
so      2a = a + N/a
        a = N/a
        a*a = N
        a = squareroot(N)
```

In fact the approximations converge rapidly to a limit. Square root programs take a tolerance (eps) and stop when two successive approximations differ by less than eps.

The algorithm is usually programmed more or less as follows:

```
C   N IS CALLED ZN HERE SO THAT IT HAS THE RIGHT TYPE
      X = A0
      Y = A0 + 2.*EPS
C   THE VALUE OF Y DOES NOT MATTER SO LONG AS ABS(X-Y).GT.EPS
100   IF (ABS(X-Y).LE.EPS) GOTO 200
      Y = X
      X = (X + ZN/X) / 2.
      GOTO 100
200   CONTINUE
C   THE SQUARE ROOT OF ZN IS NOW IN X
```

This program is indivisible in conventional languages. We will express it in a more modular form using lazy evaluation, and then show some other uses to which the parts may be put.

Since the Newton-Raphson algorithm computes a sequence of approximations it is natural to represent this explicitly in the program by a list of approximations. Each approximation is derived from the previous one by the function

```
next N x = (x + N/x) / 2
```

so (next N) is the function mapping one approximation onto the next. Calling this function f, the sequence of approximations is

```
[a0, f a0, f(f a0), f(f(f a0)), ..]
```

We can define a function to compute this:

```
repeat f a = cons a (repeat f (f a))
```

so that the list of approximations can be computed by

```
repeat (next N) a0
```

Repeat is an example of a function with an "infinite" output - but it doesn't matter, because no more approximations will actually be computed than the rest of the program requires. The infinity is only potential: all it means is that any number of approximations can be computed if required, repeat itself places no limit.

The remainder of a square root finder is a function *within*, that takes a tolerance and a list of approximations and looks down the list for two successive approximations that differ by no more than the given tolerance. It can be defined by

```
within eps (cons a (cons b rest)) =
        = b,                           if abs(a-b) <= eps
        = within eps (cons b rest), otherwise
```

Putting the parts together,

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

Now that we have the parts of a square root finder, we can try combining them in different ways. One modification we might wish to make is to wait for the ratio between successive approximations to approach one, rather than for the difference to approach zero. This is more appropriate for very small numbers (when the difference between successive approximations is small to start with) and for very large ones (when rounding error could be much larger than the tolerance). It is only necessary to define a replacement for within:

```
relative eps (cons a (cons b rest)) =
        = b,                             if abs(a-b) <= eps*abs b
        = relative eps (cons b rest), otherwise
```

Now a new version of sqrt can be defined by

```
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

It is not necessary to rewrite the part that generates approximations.

4.2 Numerical Differentiation

We have re-used the sequence of approximations to a square root. Of course, it is also possible to re-use within and relative with any numerical algorithm that generates a sequence of approximations. We will do so in a numerical differentiation algorithm.

The result of differentiating a function at a point is the slope of the function's graph at that point. It can be estimated quite easily by evaluating the function at the given point and at another point nearby and computing the slope of a straight line between the two points. This assumes that, if the two points are close enough together then the graph of the function will not curve much in between. This gives the definition

```
easydiff f x h = (f(x+h)-f x) / h
```

In order to get a good approximation the value of h should be very small. Unfortunately, if h is too small then the two values f(x+h) and f(x) are very close together, and so the rounding error in the subtraction may swamp the result. How can the right value of h be chosen? One solution to this dilemma is to compute a sequence of approximations with smaller and smaller values of h, starting with a reasonably large one. Such a sequence should converge to the value of the derivative, but will become hopelessly inaccurate eventually due to rounding error. If (within eps) is used to select the first approximation that is accurate enough then the risk of rounding error affecting the result can be much reduced. We need a function to compute the sequence:

```
differentiate h0 f x = map (easydiff f x) (repeat halve h0)
halve x = x/2
```

Here h0 is the initial value of h, and successive values are obtained by repeated halving. Given this function, the derivative at any point can be computed by

```
within eps (differentiate h0 f x)
```

Even this solution is not very satisfactory because the sequence of approximations converges fairly slowly. A little simple mathematics can help here. The elements of the sequence can be expressed as

```
the right answer + an error term involving h
```

and it can be shown theoretically that the error term is roughly proportional to a power of h, so that it gets smaller as h gets smaller. Let the right answer be A, and the error term be B*h**n. Since each approximation is computed using a value of h twice that used for the next one, any two successive approximations can be expressed as

```
        a(i)   = A + B*(2**n)*(h**n)
and     a(i+1) = A + B*(h**n)
```

Now the error term can be eliminated. We conclude

```
            a(i+1)*(2**n) - a(i)
        A = --------------------
                  2**n - 1
```

Of course, since the error term is only roughly a power of h this conclusion is also approximate, but it is a much better approximation. This improvement can be applied to all successive pairs of approximations using the function

```
elimerror n (cons a (cons b rest)) =
        = cons ((b*(2**n)-a)/(2**n-1)) (elimerror n (cons b rest))
```

Eliminating error terms from a sequence of approximations yields another sequence which converges much more rapidly.

One problem remains before we can use elimerror - we have to know the right value of n. This is difficult to predict in general, but is easy to measure. It is not difficult to show that the following function estimates it correctly, but we won't include the proof here.

```
order (cons a (cons b (cons c rest))) =
        = round(log2( (a-c)/(b-c) - 1 ))
round x = x rounded to the nearest integer
log2 x = the logarithm of x to the base 2
```

Now a general function to improve a sequence of approximations can be defined:

```
improve s = elimerror (order s) s
```

The derivative of a function f can be computed more efficiently using improve, as follows

```
within eps (improve (differentiate h0 f x))
```

Improve only works on sequences of approximations which are computed using a parameter h, which is halved between each approximation. However, if it is applied to such a sequence its result is also such a sequence! This means that a sequence of approximations can be improved more than once. A different error term is eliminated each time, and the resulting sequences converge faster and faster. So, one could compute a derivative very efficiently using

```
within eps (improve (improve (improve (differentiate h0 f x))))
```

In numerical analysts terms, this is likely to be a fourth order method, and gives an accurate result very quickly. One could even define

```
super s = map second (repeat improve s)
second (cons a (cons b rest)) = b
```

which uses repeat improve to get a sequence of more and more improved sequences of approximations, and constructs a new sequence of approximations by taking the second approximation from each of the improved sequences (it turns out that the second one is the best one to take - it is more accurate than the first and doesn't require any extra work to compute). This algorithm is really very sophisticated - it uses a better and better numerical method as more and more approximations are computed. One could compute derivatives very efficiently indeed with the program:

```
within eps (super (differentiate h0 f x))
```

This is probably a case of using a sledge-hammer to crack a nut, but the point is that even an algorithm as sophisticated as super is easily expressed when modularised using lazy evaluation.

4.3 Numerical Integration

The last example we will discuss in this section is numerical integration. The problem may be stated very simply: given a real valued function f of one real argument, and two end-points a and b, estimate the area under the curve f describes between the end-points. The easiest way to estimate the area is to assume that f is nearly a straight line, in which case the area would be

```
easyintegrate f a b = (f a + f b)*(b-a)/2
```

Unfortunately this estimate is likely to be very inaccurate unless a and b are close together. A better estimate can be made by dividing the interval from a to b in two, estimating the area on each half, and adding the results together. We can define a sequence of better and better approximations to the value of the integral by using the formula above for the first approximation, and then adding together better and better approximations to the integrals on each half to calculate the others. This sequence is computed by the function

```
integrate f a b = cons (easyintegrate f a b)
                       (map addpair (zip (integrate f a mid)
                                         (integrate f mid b)))
        where mid = (a+b)/2
```

Zip is another standard list-processing function. It takes two lists and returns a list of pairs, each pair consisting of corresponding elements of the two lists. Thus the first pair consists of the first element of the first list and the first element of the second, and so on. Zip can be defined by

```
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```

In integrate, zip computes a list of pairs of corresponding approximations to the integrals on the two sub-intervals, and map addpair adds the elements of the pairs together to give a list of approximations to the original integral.

Actually, this version of integrate is rather inefficient because it continually recomputes values of f. As written, easyintegrate evaluates f at a and at b, and then the recursive calls of integrate re-evaluate each of these. Also, (f mid) is evaluated in each recursive call. It is therefore preferable to use the following version which never recomputes a value of f.

```
integrate f a b = integ f a b (f a) (f b)
integ f a b fa fb = cons ((fa+fb)*(b-a)/2)
                         (map addpair (zip (integ f a m fa fm)
                                           (integ f m b fm fb)))
        where m = (a+b)/2
              fm = f m
```

Integrate computes an infinite list of better and better approximations to the integral, just as differentiate did in the section above. One can therefore just write down integration routines that integrate to any required accuracy, as in

```
within eps (integrate f a b)
relative eps (integrate f a b)
```

This integration algorithm suffers from the same disadvantage as the first differentiation algorithm in the preceding sub-section - it converges rather slowly. Once again, it can be improved. The first approximation in the sequence is computed (by easyintegrate) using only two points, with a separation of b-a. The second approximation also uses the mid-point, so that the separation between neighbouring points is only (b-a)/2. The third approximation uses this method on each half-interval, so the separation between neighbouring points is only (b-a)/4. Clearly the separation between neighbouring points is halved between each approximation and the next. Taking this separation as “h”, the sequence is a candidate for improvement using the “improve” function defined in the preceding section. Therefore we can now write down quickly converging sequences of approximations to integrals, for example

```
super (integrate sin 0 4)
improve (integrate f 0 1)
where f x = 1/(1+x*x)
```

(This latter sequence is an eighth order method for computing pi/4. The second approximation, which requires only five evaluations of f to compute, is correct to five decimal places).

In this section we have taken a number of numerical algorithms and programmed them functionally, using lazy evaluation as glue to stick their parts together. Thanks to this, we were able to modularise them in new ways, into generally useful functions such as within, relative and improve. By combining these parts in various ways we programmed some quite good numerical algorithms very simply and easily.

5 An Example from Artificial Intelligence

We have argued that functional languages are powerful primarily because they provide two new kinds of glue: higher-order functions and lazy evaluation. In this section we take a larger example from Artificial Intelligence and show how it can be programmed quite simply using these two kinds of glue.

The example we choose is the alpha-beta "heuristic", an algorithm for estimating how good a position a game-player is in. The algorithm works by looking ahead to see how the game might develop, but avoids pursuing unprofitable lines.

Let game-positions be represented by objects of the type "position". This type will vary from game to game, and we assume nothing about it. There must be some way of knowing what moves can be made from a position: assume that there is a function

```
moves: position -> listof position
```

that takes a game-position as its argument and returns the list of all positions that can be reached from it in one move. Taking noughts and crosses (tic-tac-toe) as an example,

```
         | |         X| |     |X|     | |
        -+-+-       -+-+-   -+-+-   -+-+-
moves    | |    =  [ | |  ,  | |  ,  |X| ]
        -+-+-       -+-+-   -+-+-   -+-+-
         | |         | |     | |     | |

         | |         O| |     |O|
        -+-+-       -+-+-   -+-+-
moves    |X|    =  [ |X|  ,  |X| ]
        -+-+-       -+-+-   -+-+-
         | |         | |     | |
```

This assumes that it is always possible to tell which player's turn it is from a position. In noughts and crosses this can be done by counting the noughts and crosses, in a game like chess one would have to include the information explicitly in the type "position".

Given the function moves, the first step is to build a game tree. This is a tree in which the nodes are labelled by positions, such that the children of a node are labelled with the positions that can be reached in one move from that node. That is, if a node is labelled with position p, then its children are labelled with the positions in (moves p). A game tree may very well be infinite, if it is possible for a game to go on for ever with neither side winning. Game trees are exactly like the trees we discussed in section 2 - each node has a label (the position it represents) and a list of subnodes. We can therefore use the same datatype to represent them.

A game tree is built by repeated applications of moves. Starting from the root position, moves is used to generate the labels for the sub-trees of the root. Moves is then used again to generate the sub-trees of the sub-trees and so on. This pattern of recursion can be expressed as a higher-order function,

```
reptree f a = node a (map (reptree f) (f a))
```

Using this function another can be defined which constructs a game tree from a particular position

```
gametree p = reptree moves p
```

For an example, look at figure 1. The higher-order function used here (reptree) is analogous to the function repeat used to construct infinite lists in the preceding section.

The alpha-beta algorithm looks ahead from a given position to see whether the game will develop favourably or unfavourably, but in order to do so it must be able to make a rough estimate of the value of a position without looking ahead. This "static evaluation" must be used at the

```
          | |
         -+-+-
gametree  | |
         -+-+-
          | |

                          | |
                         -+-+-
=                         | |
                         -+-+-
                          | |
                        /  |  \
                       /   |   \
                      /    |    \
                     /     |     \
                    /      |      \
                   /       |       \
               X| |       |X|       | |
               -+-+-     -+-+-     -+-+-
                | |       | |       |X|
               -+-+-     -+-+-     -+-+-
                | |       | |       | |
                /|\       /|\        /\
                ...       ...       /  \
                                   /    \
                                  /      \
                             O| |        |O|
                             -+-+-      -+-+-
                              |X|        |X|
                             -+-+-      -+-+-
                              | |        | |
                              /|\        /|\
                              ...        ...
```

Figure 1: An Example of a Game-Tree.

limit of the look-ahead, and may be used to guide the algorithm earlier. The result of the static evaluation is a measure of the promise of a position from the computer's point of view (assuming that the computer is playing the game against a human opponent). The larger the result, the better the position for the computer. The smaller the result, the worse the position. The simplest such function would return (say) +1 for positions where the computer has already won, -1 for positions where the computer has already lost, and 0 otherwise. In reality, the static evaluation function measures various things that make a position "look good", for example material advantage and control of the centre in chess. Assume that we have such a function,

```
static: position -> number
```

Since a game-tree is a (treeof position), it can be converted into a (treeof number) by the function (maptree static), which statically evaluates all the positions in the tree (which may be infinitely many). This uses the function maptree defined in section 2.

Given such a tree of static evaluations, what is the true value of the positions in it? In particular, what value should be ascribed to the root position? Not its static value, since this is only a rough guess. The value ascribed to a node must be determined from the true values of its subnodes. This can be done by assuming that each player makes the best moves he can. Remembering that a high value means a good position for the computer, it is clear that when it is the computer's move from any position, it will choose the move leading to the sub-node with the maximum true value. Similarly, the opponent will choose the move leading to the sub-node with the minimum true value. Assuming that the computer and its opponent alternate turns, the true value of a node is computed by the function maximise if it is the computer's turn and minimise if it is not:

```
maximise (node n sub) = max (map minimise sub)
minimise (node n sub) = min (map maximise sub)
```

Here max and min are functions on lists of numbers that return the maximum and minimum of the list respectively. These definitions are not complete because they recurse for ever - there is no base case. We must define the value of a node with no successors, and we take it to be the static evaluation of the node (its label). Therefore the static evaluation is used when either player has already won, or at the limit of look-ahead. The complete definitions of maximise and minimise are

```
maximise (node n nil) = n
maximise (node n sub) = max (map minimise sub)
minimise (node n nil) = n
minimise (node n sub) = min (map maximise sub)
```

One could almost write down a function at this stage that would take a position and return its true value. This would be:

```
evaluate = maximise . maptree static . gametree
```

There are two problems with this definition. First of all, it doesn't work for infinite trees. Maximise keeps on recursing until it finds a node with no subtrees - an end to the tree. If there is no end then maximise will return no result. The second problem is related - even finite game trees (like the one for noughts and crosses) can be very large indeed. It is unrealistic to try to evaluate the whole of the game tree - the search must be limited to the next few moves. This can be done by pruning the tree to a fixed depth,

```
prune 0 (node a x) = node a nil
prune n (node a x) = node a (map (prune (n-1)) x)
```

(prune n) takes a tree and "cuts off" all nodes further than n from the root. If a game tree is pruned it forces maximise to use the static evaluation for nodes at depth n, instead of recursing further. Evaluate can therefore be defined by

```
evaluate = maximise . maptree static . prune 5 . gametree
```

which looks (say) 5 moves ahead.

Already in this development we have used higher-order functions and lazy evaluation. Higher order functions reptree and maptree allow us to construct and manipulate game trees with ease. More importantly, lazy evaluation permits us to modularise evaluate in this way. Since gametree has a potentially infinite result, this program would never terminate without lazy evaluation. Instead of writing

```
prune 5 . gametree
```

we would have to fold these two functions together into one which only constructed the first five levels of the tree. Worse, even the first five levels may be too large to be held in memory at one time. In the program we have written, the function

```
maptree static . prune 5 . gametree
```

only constructs parts of the tree as maximise requires them. Since each part can be thrown away (reclaimed by the garbage collector) as soon as maximise has finished with it, the whole tree is never resident in memory. Only a small part of the tree is stored at a time. The lazy program is therefore efficient. Since this efficiency depends on an interaction between maximise (the last function in the chain of compositions) and gametree (the first), it could only be achieved without lazy evaluation by folding all the functions in the chain together into one big one. This is a drastic reduction in modularity, but it is what is usually done. We can make improvements to this evaluation algorithm by tinkering with each part: this is relatively easy. A conventional programmer must modify the entire program as a unit, which is much harder.

So far we have only described simple minimaxing. The heart of the alpha-beta algorithm is the observation that one can often compute the value of maximise or minimise without looking at the whole tree. Consider the tree:

```
             max
             / \
            /   \
           /     \
          /       \
        min       min
        / \       / \
       /   \     /   \
      1     2   0     ?
```

Strangely enough, it is unnecessary to know the value of the question mark in order to evaluate the tree. The left minimum evaluates to 1, but the right minimum clearly evaluates to something less than or equal to 0. Therefore the maximum of the two minima must be 1. This observation can be generalised and built into maximise and minimise.

The first step is to separate maximise into an application of max to a list of numbers; that is, we decompose maximise as

```
maximise = max . maximise'
```

(Minimise is decomposed in a similar way. Since minimise and maximise are entirely symmetrical we shall discuss maximise and assume that minimise is treated similarly). Once decomposed in this way, maximise can use minimise' rather than minimise itself, to discover which numbers minimise would take the minimum of. It may then be able to discard some of the numbers without looking at them. Thanks to lazy evaluation, if maximise doesn't look at all of the list of numbers, some of them will not be computed, with a potential saving in computer time.

It is easy to "factor out" max from the definition of maximise, giving

```
maximise' (node n nil) = cons n nil
maximise' (node n l) = map minimise l
                     = map (min . minimise') l
                     = map min (map minimise' l)
                     = mapmin (map minimise' l)
where mapmin = map min
```

Since minimise' returns a list of numbers, the minimum of which is the result of minimise, (map minimise' l) returns a list of lists of numbers. Maximise' should return a list of the minima of those lists. However, only the maximum of this list matters. We shall define a new version of mapmin which omits the minima of lists whose minimum doesn't matter.

```
mapmin (cons nums rest) =
        = cons (min nums) (omit (min nums) rest)
```

The function omit is passed a "potential maximum" - the largest minimum seen so far - and omits any minima which are less than this.

```
omit pot nil = nil
omit pot (cons nums rest) =
        = omit pot rest,                          if minleq nums pot
        = cons (min nums) (omit (min nums) rest), otherwise
```

Minleq takes a list of numbers and a potential maximum, and returns true if the minimum of the list of numbers is less than or equal to the potential maximum. To do this, it does not need to look at all the list! If there is any element in the list less than or equal to the potential maximum, then the minimum of the list is sure to be. All elements after this particular one are irrelevant - they are like the question mark in the example above. Therefore minleq can be defined by

```
minleq nil pot = false
minleq (cons num rest) pot = true,            if num<=pot
                           = minleq rest pot, otherwise
```

Having defined maximise' and minimise' in this way it is simple to write a new evaluator:

```
evaluate = max . maximise' . maptree static . prune 8 . gametree
```

Thanks to lazy evaluation, the fact that maximise' looks at less of the tree means that the whole program runs more efficiently, just as the fact that prune looks at only part of an infinite tree enables the program to terminate. The optimisations in maximise', although fairly simple, can have a dramatic effect on the speed of evaluation, and so can allow the evaluator to look further ahead.

Other optimisations can be made to the evaluator. For example, the alpha-beta algorithm just described works best if the best moves are considered first, since if one has found a very good move then there is no need to consider worse moves, other than to demonstrate that the opponent has at least one good reply to them. One might therefore wish to sort the sub-trees at each node, putting those with the highest values first when it is the computer's move, and those with the lowest values first when it is not. This can be done with the function

```
highfirst (node n sub) = node n (sort higher (map lowfirst sub))
lowfirst (node n sub) = node n (sort (not.higher) (map highfirst sub))
higher (node n1 sub1) (node n2 sub2) = n1>n2
```

where sort is a general purpose sorting function. The evaluator would now be defined by

```
evaluate = max . maximise' . highfirst . maptree static .
                prune 8 . gametree
```

One might regard it as sufficient to consider only the three best moves for the computer or the opponent, in order to restrict the search. To program this, it is only necessary to replace highfirst with (taketree 3 . highfirst), where

```
taketree n = redtree (nodett n) cons nil
nodett n label sub = node label (take n sub)
```

Taketree replaces all the nodes in a tree with nodes with at most n subnodes, using the function (take n) which returns the first n elements of a list (or fewer if the list is shorter than n).

Another improvement is to refine the pruning. The program above looks ahead a fixed depth even if the position is very dynamic - it may decide to look no further than a position in which the queen is threated in chess, for example. It is usual to define certain "dynamic" positions and not to allow look-ahead to stop in one of these. Assuming a function "dynamic" that recognises such positions, we need only add one equation to prune to do this:

```
prune 0 (node pos sub) = node pos (map (prune 0) sub),
                                          if dynamic pos
```

Making such changes is easy in a program as modular as this one. As we remarked above, since the program depends crucially for its efficiency on an interaction between maximise, the last function in the chain, and gametree, the first, it can only be written as a monolithic program without lazy evaluation. Such a program is hard to write, hard to modify, and very hard to understand.

6 Conclusion

In this paper, we've argued that modularity is the key to successful programming. Languages which aim to improve productivity must support modular programming well. But new scope rules and mechanisms for separate compilation are not enough - modularity means more than modules. Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To assist modular programming, a language must provide good glue. Functional programming languages provide two new kinds of glue - higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways, and we've shown many examples of this. Smaller and more general modules can be re-used more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones. It also provides a target for functional programmers to aim at. If any part of a program is messy or complicated, the programmer should attempt to modularise it and to generalise the parts. He should expect to use higher-order functions and lazy evaluation as his tools for doing this.

Of course, we are not the first to point out the power and elegance of higher-order functions and lazy evaluation. For example, Turner shows how both can be used to great advantage in a program for generating chemical structures [Tur81]. Abelson and Sussman stress that streams (lazy lists) are a powerful tool for structuring programs [AS86]. Henderson has used streams to structure functional operating systems [P.H82]. The main contribution of this paper is to assert that better modularity alone is the key to the power of functional languages.

It is also relevant to the present controversy over lazy evaluation. Some believe that functional languages should be lazy, others believe they should not. Some compromise and provide only lazy lists, with a special syntax for constructing them (as, for example, in SCHEME [AS86]). This paper provides further evidence that lazy evaluation is too important to be relegated to second-class citizenship. It is perhaps the most powerful glue functional programmers possess. One should not obstruct access to such a vital tool.

Acknowledgements

This paper owes much to many conversations with Phil Wadler and Richard Bird in the Programming Research Group at Oxford. Magnus Bondesson at Chalmers University, Goteborg pointed out a serious error in an earlier version of one of the numerical algorithms, and thereby prompted development of many of the others. This work was carried out with the support of a Research Fellowship from the UK Science and Engineering Research Council.

References

[AS86] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Boston, 1986.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2), 1989.

[Hug90] John Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming.* Addison Wesley, 1990.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[oD80] United States Department of Defense. *The Programming Language Ada Reference Manual.* Springer-Verlag, 1980.

[P.H82] P.Henderson. Purely Functional Operating Systems. 1982.

[Tur81] D. A. Turner. The Semantic Elegance of Applicative Languages. In *Proceedings 1981 Conference on Functional Languages and Computer Architecture*, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981.

[Tur85] D. A. Turner. Miranda: A non-strict language with polymorphic types. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, 1985.

[Wir82] N. Wirth. *Programming in Modula-II.* Springer-Verlag, 1982.