*From "Research Topics in Functional Programming" ed. D. Turner, Addison-Wesley, 1990, pp 17–42.*

# Why Functional Programming Matters

John Hughes
The University, Glasgow

**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 and to debug, and provides a collection of modules that can be reused to reduce future programming costs. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute significantly 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). We conclude that since modularity is the key to successful programming, functional programming offers important advantages for software development.

## 1 Introduction

This paper is an attempt to demonstrate to the larger community of (non-functional) programmers the significance of functional programming, 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 its fundamental operation is the application of functions to arguments. A main program itself is written as a function that 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. All of these functions are much like ordinary mathematical functions, and in this paper they will be

[1]An earlier version of this paper appeared in the *The Computer Journal*, 32(2):98–107, April 1989. 

defined by ordinary equations. We are following Turner's language Miranda[4][2] here, but the notation should be readable without specific knowledge of this.

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 an expression's value, 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 isn't (it has no assignment, no side effects, no flow of control) but not much about what it is. The functional programmer sounds rather like a mediæval 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 totally unconvincing.

Functional programmers argue that there *are* great material benefits — that a functional programmer is an order of magnitude more productive than his or her 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 no help in exploiting the power of functional languages. One cannot write a program that 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 characterization of functional programming is inadequate. We must find something to put in its place — something that not only explains the power of functional programming but also gives a clear indication of what the functional programmer should strive towards.

[2]Miranda is a trademark of Research Software Ltd.

## 2 An Analogy with Structured Programming

It's 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 *goto*s" and so on.

With the benefit of hindsight, it's 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. Second, general-purpose modules can be reused, leading to faster development of subsequent programs. Third, the modules of a program can be tested independently, helping to reduce the time spent debugging.

The absence of *goto*s, 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 recent languages such as MODULA-II [6] and Ada [5] 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 subproblems, then solves the subproblems, and finally 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 one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language. Complicated scope rules and provision for separate compilation help only with clerical details — they can never make a great contribution to modularization.

We shall argue in the remainder of this paper that functional languages provide two new, very important kinds of glue. We shall give some examples of programs that can be modularized in new ways and can thereby be simplified. This is the key to functional programming's power — it allows improved modularization. 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 Gluing 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 the elements of a list. We can define lists[3] by

$$listof * ::= Nil \mid Cons * (listof *)$$

which means that a list of $*$s (whatever $*$ is) is either *Nil*, representing a list with no elements, or a *Cons* of a $*$ and another list of $*$s. A *Cons* represents a list whose first element is the $*$ and whose second and subsequent elements are the elements of the other list of $*$s. Here $*$ may stand for any type — for example, if $*$ 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 *Cons*es and *Nil*s 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 by a recursive function *sum*. The function *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

$$sum\ (Cons\ n\ list) = num + sum\ list$$

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

$$sum\ Nil\ = \boxed{0}$$
$$sum\ (Cons\ n\ list) = n\ \boxed{+}\ sum\ list$$

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

$$sum\ =\ foldr\ (+)\ 0$$

[3] In Miranda, lists can also be defined using the built-in constructor (:), but the notation used here is equally valid.

The definition of *foldr* can be derived just by parameterizing the definition of *sum*, giving

$$(foldr\ f\ x)\ Nil\ = x$$
$$(foldr\ f\ x)\ (Cons\ a\ l\ ) = f\ a\ ((foldr\ f\ x)\ l\ )$$

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

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

$$product\ =\ foldr\ (*)\ 1$$

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

$$anytrue\ =\ foldr\ (\vee)\ False$$

or whether they are all true

$$alltrue\ =\ foldr\ (\wedge)\ True$$

One way to understand $(foldr\ f\ a)$ 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 $(foldr\ (+)\ 0)$ converts it into

$$(+)\ 1\ ((+)\ 2\ ((+)\ 3\ 0)) = 6$$

and $(foldr\ (*)\ 1)$ converts it into

$$(*)\ 1\ ((*)\ 2\ ((*)\ 3\ 1)) = 6$$

Now it's obvious that $(foldr\ Cons\ Nil)$ just copies a list. Since one list can be appended to another by *Cons*ing its elements onto the front, we find

$$append\ a\ b =\ foldr\ Cons\ b\ a$$

As an example,

$$\begin{aligned} append\ [1,2]\ [3,4] &= foldr\ Cons\ [3,4]\ [1,2] \\ &= foldr\ Cons\ [3,4]\ (Cons\ 1\ (Cons\ 2\ Nil)) \\ &= Cons\ 1\ (Cons\ 2\ [3,4])) \\ &\qquad \text{(replacing } Cons \text{ by } Cons \text{ and } Nil \text{ by } [3,4]) \\ &= [1,2,3,4] \end{aligned}$$

We can count the number of elements in a list using the function *length*, defined by

$$length \;=\; foldr\, count\; 0$$
$$count\; a\; n \;=\; n+1$$

because *count* increments 0 as many times as there are *Cons*es. A function that doubles all the elements of a list could be written as

$$doubleall \;=\; foldr\, doubleandcons\; Nil$$

where

$$doubleandcons\; n\; list \;=\; Cons\; (2*n)\; list$$

The function *doubleandcons* can be modularized 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:

$$\begin{aligned} fandcons\; f\; el \;&=\; (Cons\; .\; f\;)\; el \\ &=\; Cons\; (f\; el\;) \end{aligned}$$

so

$$fandcons\; f\; el\; list = \; Cons\; (f\; el\;)\; list$$

The final version is

$$doubleall \;= foldr\; (Cons\; .\; double)\; Nil$$

With one further modularization we arrive at

$$doubleall \;= map\; double$$
$$map\; f = foldr\; (Cons\; .\; f\;)\; Nil$$

where *map* — another generally useful function — applies any function $f$ to all the elements of a list.

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

$$summatrix \;=\; sum \;.\; map\; sum$$

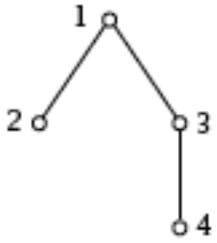
The function *map sum* uses *sum* to add up all the rows, and then the leftmost *sum* adds up the row totals to get the sum of the whole matrix.

These examples should be enough to convince the reader that a little modularization can go a long way. By modularizing a simple function (*sum*) as a combination of a "higher-order function" and some simple arguments, we have arrived at a part ($foldr$) that can be used to write 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 labeled trees, defined by

$$treeof\; * ::= Node\; *\; (listof\; (treeof\; *))$$

This definition says that a tree of $*$s is a node, with a label which is a $*$, and a list of subtrees which are also trees of $*$s. For example, the tree




would be represented by

$$\begin{array}{l} Node\; 1 \\ \quad (Cons\; (Node\; 2\; Nil) \\ \qquad (Cons\; (Node\; 3 \\ \qquad\qquad (Cons\; (Node\; 4\; Nil)\; Nil)) \\ \qquad\quad Nil)) \end{array}$$

Instead of considering an example and abstracting a higher-order function from it, we will go straight to a function *foldtree* analogous to *foldr*. Recall that *foldr* took two arguments: something to replace *Cons* with and something to replace *Nil* with. Since trees are built using *Node*, *Cons*, and *Nil*, *foldtree* must take three arguments — something to replace each of these with. Therefore we define

$$\begin{array}{l} foldtree\; f\; g\; a\; (Node\; label\; subtrees) = \\ \qquad f\; label\; (foldtree\; f\; g\; a\; subtrees) \\ foldtree\; f\; g\; a\; (Cons\; subtree\; rest) = \\ \qquad g\; (foldtree\; f\; g\; a\; subtree)\; (foldtree\; f\; g\; a\; rest) \\ foldtree\; f\; g\; a\; Nil\; = a \end{array}$$

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

$$sumtree \;=\; foldtree\,(+)\,(+)\,0$$

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

$$\begin{array}{l}(+)\;\;1\\ \qquad((+)\;((+)\;2\;0)\\ \qquad\qquad((+)\;((+)\;3\\ \qquad\qquad\qquad\qquad((+)\;((+)\;4\;0)\;0))\\ \qquad\qquad\quad 0))\\ =10\end{array}$$

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

$$labels \;=\; foldtree\;Cons\;append\;Nil$$

The same example gives

$$\begin{array}{l}Cons\;1\\ \qquad(append\;\;(Cons\;2\;\;Nil)\\ \qquad\qquad\quad(append\;(Cons\;3\\ \qquad\qquad\qquad\qquad\qquad(append\;(Cons\;4\;Nil)\;Nil))\\ \qquad\qquad\qquad\quad Nil))\\ =[1,2,3,4]\end{array}$$

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

$$maptree\;f \;=\; foldtree\;(Node\;.\;f\;)\;Cons\;Nil$$

All this can be achieved because functional languages allow functions that are indivisible in conventional programming languages to be expressed as a combinations of parts — a general higher-order function and some particular specializing 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 it also localizes knowledge about the details of its representation. The best analogy with conventional programming is with extensible languages — in effect, the programming language can be extended with new control structures whenever desired.

# 4 Gluing 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 that, when applied to its input, computes

$$g\ (f\ \text{input})$$

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 synchronization. Program $f$ is started only when $g$ tries to read some input, and runs only 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. Program $f$ can even be a nonterminating 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 modularization.

Since this method of evaluation runs $f$ as little as possible, it is called "lazy evaluation". It makes it practical to modularize a program as a generator that constructs a large number of possible answers, and a selector that chooses the appropriate one. While some other systems allow programs to be run together in this manner, only functional languages (and not even all of them) use lazy evaluation uniformly for every function call, allowing any part of a program to be modularized in this way. Lazy evaluation is perhaps the most powerful tool for modularization in the functional programmer's repertoire.

We have described lazy evaluation in the context of functional languages, but surely so useful a feature should be added to nonfunctional languages — or should it? Can lazy evaluation and side-effects coexist? Unfortunately, they cannot: Adding lazy evaluation to an imperative notation is not actually impossible, but the combination would make the programmer's life harder, rather than easier. Because lazy evaluation's power depends on the programmer giving up any direct control over the order in which the parts of a program are executed, it would make programming with side effects rather difficult, because predicting in what order —or even whether— they might take place would require knowing a lot about the context in which they are embedded. Such global interdependence would defeat the very modularity that —in functional languages— lazy evaluation is designed to enhance.

## 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

$$a_{i+1} = (a_i + n/a_i)/2$$

If the approximations converge to some limit $a$, then

$$a = (a + n/a)/2$$

so

$$\begin{aligned} 2a &= a + n/a \\ a &= n/a \\ a * a &= n \\ a &= \sqrt{n} \end{aligned}$$

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   Y'S VALUE 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

$$[a0,\ f\ a0,\ f\ (f\ a0),\ f\ (f\ (f\ a0)),\ \ldots\ ]$$

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

$$repeat\ (next\ n)\ a0$$

The function *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*, which 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

$$\begin{aligned} &within\ eps\ (Cons\ a\ (Cons\ b\ rest)) \\ &\quad = b, && \textbf{if } abs\ (a-b) \leq\ eps \\ &\quad = within\ eps\ (Cons\ b\ rest), && \textbf{otherwise} \end{aligned}$$

Putting the parts together, we have

$$sqrt\ a0\ eps\ n =\ within\ eps\ (repeat\ (next\ n)\ a0)$$

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 1, rather than for the difference to approach 0. 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*:

$$\begin{aligned} &relative\ eps\ (Cons\ a\ (Cons\ b\ rest)) \\ &\quad = b, && \textbf{if } abs\ (a/b-1) \leq\ eps \\ &\quad = relative\ eps\ (Cons\ b\ rest), && \textbf{otherwise} \end{aligned}$$

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 reused the sequence of approximations to a square root. Of course, it is also possible to reuse *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

$$\mathit{easydiff}\ f\ x\ h = (f(x+h) - f\ x)/h$$

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:

$$\mathit{differentiate}\ h0\ f\ x = \mathit{map}\ (\mathit{easydiff}\ f\ x)\ (\mathit{repeat}\ \mathit{halve}\ h0)$$
$$\mathit{halve}\ x = x/2$$

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

$$\mathit{within}\ \mathit{eps}\ (\mathit{differentiate}\ h0\ f\ x)$$

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 let the error term be $B \times h^n$. 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

$$a_i = A + B \times 2^n \times h^n$$

and

$$a_{i+1} = A + B \times h^n$$

Now the error term can be eliminated. We conclude

$$A = \frac{a_{n+1} \times 2^n - a_n}{2^n - 1}$$

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

$$\begin{aligned}&\mathit{elimerror}\ n\ (\mathit{Cons}\ a\ (\mathit{Cons}\ b\ \mathit{rest}))\\ &=\ \mathit{Cons}\ ((b*(2\hat{}n)-a)/(2\hat{}n-1))\ (\mathit{elimerror}\ n\ (\mathit{Cons}\ b\ \mathit{rest}))\end{aligned}$$

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's not difficult to show that the following function estimates it correctly, but we won't include the proof here:

$$\begin{aligned}&\mathit{order}\ (\mathit{Cons}\ a\ (\mathit{Cons}\ b\ (\mathit{Cons}\ c\ \mathit{rest})))\\ &\qquad =\ \mathit{round}\ (\mathit{log}2\ ((a-c)/(b-c)-1))\\ &\mathit{round}\ x = x \text{ rounded to the nearest integer}\\ &\mathit{log}2\ x =\ \text{the logarithm of } x \text{ to the base 2}\end{aligned}$$

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

$$\mathit{improve}\ s =\ \mathit{elimerror}\ (\mathit{order}\ s)\ s$$

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

$$\mathit{within}\ \mathit{eps}\ (\mathit{improve}\ (\mathit{differentiate}\ h0\ f\ x))$$

The function *improve* works only on sequences of approximations that are computed using a parameter $h$, which is halved for each successive 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. Hence one could compute a derivative very efficiently using

$$\mathit{within}\ \mathit{eps}\ (\mathit{improve}\ (\mathit{improve}\ (\mathit{improve}\ (\mathit{differentiate}\ h0\ f\ x))))$$

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

$$\begin{aligned}&\mathit{super}\ s =\ \mathit{map}\ \mathit{second}\ (\mathit{repeat}\ \mathit{improve}\ s)\\ &\mathit{second}\ (\mathit{Cons}\ a\ (\mathit{Cons}\ b\ \mathit{rest})) = b\end{aligned}$$

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 sledgehammer to crack a nut, but the point is that even an algorithm as sophisticated as *super* is easily expressed when modularized 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 points $a$ and $b$, estimate the area under the curve that $f$ describes between the 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. 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

$$\begin{aligned} integrate\ f\ a\ b\ = Cons\ &(easyintegrate\ f\ a\ b) \\ &(map\ addpair\ (zip2\ (integrate\ f\ a\ mid\ ) \\ &\qquad (integrate\ f\ mid\ b))) \\ \textbf{where}\ mid\ &= (a + b)/2 \end{aligned}$$

The function *zip*2 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. We can define *zip*2 by

$$zip2\ (Cons\ a\ s)\ (Cons\ b\ t) = Cons\ (a, b)\ (zip2\ s\ t)$$

In *integrate*, *zip*2 computes a list of pairs of corresponding approximations to the integrals on the two subintervals, 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 $f$:

$$
\begin{aligned}
&integrate\ f\ a\ b = \ integ\ f\ a\ b\ (f\ a)\ (f\ b)\\
&integ\ f\ a\ b\ fa\ fb\ = \quad Cons\ \ ((fa+fb)*(b-a)/2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad map\ addpair(zip2\ \ (integ\ f\ a\ m\ fa\ fm)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (integ\ f\ m\ b\ fm\ fb)))\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{where}\quad m=(a+b)/2\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ fm=f\ m
\end{aligned}
$$

The function *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

$$
\begin{aligned}
&within\ eps\ (integrate\ f\ a\ b)\\
&relative\ eps\ (integrate\ f\ a\ b)
\end{aligned}
$$

This integration algorithm suffers from the same disadvantage as the first differentiation algorithm in the preceding subsection — 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 midpoint, so that the separation between neighboring points is only $(b-a)/2$. The third approximation uses this method on each half-interval, so the separation between neighboring points is only $(b-a)/4$. Clearly the separation between neighboring points is halved between each approximation and the next. Taking this separation as $h$, the sequence is a candidate for improvement using the function *improve* defined in the preceding section. Therefore we can now write down quickly converging sequences of approximations to integrals, for example,

$$super\ (integrate\ sin\ 0\ 4)$$

and

$$
\begin{aligned}
&improve\ (integrate\ f\ 0\ 1)\\
&\textbf{where}\ f\ x = 1/(1+x*x)
\end{aligned}
$$

(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 have been able to modularize them in new ways, into generally useful functions such as *within*, *relative*, and *improve*. By combining these parts in various ways we have 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 it 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,

$$moves :: position \rightarrow listof\ position$$

that takes a game-position as its argument and returns the list of all positions that can be reached from it in one move. As an example, Fig. 1 shows *moves* for a couple of positions in tic-tac-toe (noughts and crosses). This assumes that it is always possible to tell which player's turn it is from a position. In tic-tac-toe this can be done by counting the ×s and Os; 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 labeled by positions, so that the children of a node are labeled with the positions that can be reached in one move from that node. That is, if a node is labeled with position $p$, then its children are labeled with the positions in $(moves\ p)$. Game trees are not all finite: If it's possible for a game to go on forever with neither side winning, its game tree is infinite. Game trees are exactly like the trees we discussed in Section 2 — each node has a

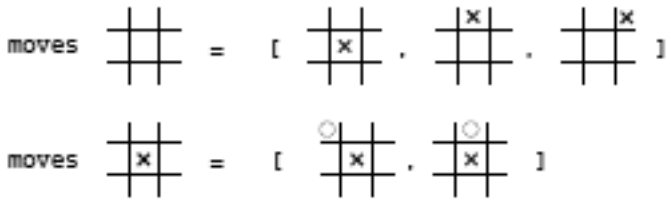



Figure 1: *moves* for two positions in tic-tac-toe.

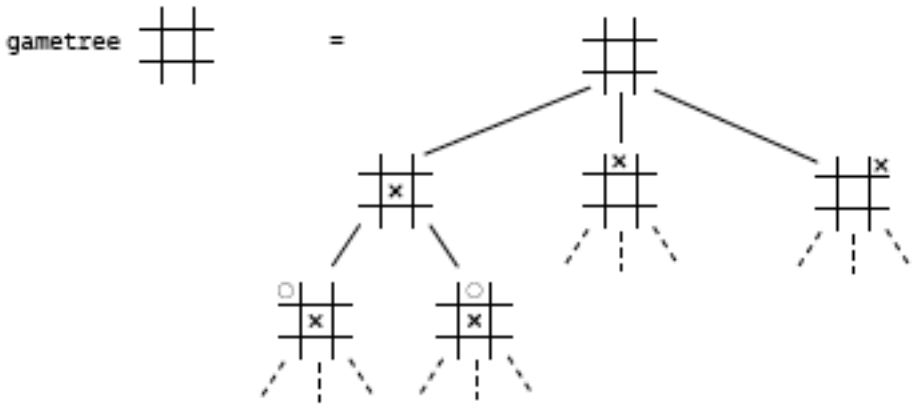



Figure 2: Part of a game tree for tic-tac-toe.

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 subtrees of the root. It is then used again to generate the subtrees of the subtrees 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:

$$gametree\ p = \ reptree\ moves\ p$$

As an example, consider Fig. 2. 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 favorably or unfavorably, 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 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 center in chess. Assume that we have such a function,

$$static :: position \rightarrow number$$

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 possible. 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 subnode with the maximum true value. Similarly, the opponent will choose the move leading to the subnode 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 *maximize* if it is the computer's turn and *minimize* if it is not:

$$maximize\ (Node\ n\ sub) = max\ (map\ minimize\ sub)$$
$$minimize\ (Node\ n\ sub) = min\ (map\ maximize\ sub)$$

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 forever — 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 *maximize* and *minimize* are

$$maximize\ (Node\ n\ Nil) = n$$
$$maximize\ (Node\ n\ sub) = max\ (map\ minimize\ sub)$$
$$minimize\ (Node\ n\ Nil) = n$$
$$minimize\ (Node\ n\ sub) = max\ (map\ maximize\ sub)$$

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

$$evaluate = maximize\ .\ maptree\ static\ .\ gametree$$

There are two problems with this definition. First of all, it doesn't work for infinite trees, because *maximize* keeps on recursing until it finds a node with no subtrees — an end to the tree. If there is no end then *maximize* will return no result. The second problem is related — even finite game trees (like the one for tic-tac-toe) 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,

$$\begin{aligned} prune\ \ 0\ \ (Node\ a\ x) &= Node\ a\ Nil \\ prune\ (n+1)\ (Node\ a\ x) &= Node\ a\ (map\ (prune\ n)\ x) \end{aligned}$$

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

$$evaluate\ =\ maximize\ .\ maptree\ static\ .\ prune\ 5\ .\ gametree$$

which looks (say) five 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 modularize *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 that constructed only 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

$$maptree\ static\ .\ prune\ 5\ .\ gametree$$

constructs parts of the tree only as *maximize* requires them. Since each part can be thrown away (reclaimed by the garbage collector) as soon as *maximize* 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. This efficiency depends on an interaction between *maximize* (the last function in the chain of compositions) and *gametree* (the first); without lazy evaluation, therefore, it could be achieved only by folding all the functions in the chain together into one big one. This would be 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 described only simple minimaxing. The heart of the alpha-beta algorithm is the observation that one can often compute the value returned by *maximize* or *minimize* without looking at the whole tree. Consider the tree:


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 at most 0. Therefore the maximum of the two minima must be 1. This observation can be generalized and built into *maximize* and *minimize*.

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

$$maximize = max\ .\ maximize'$$

(We decompose *minimize* in a similar way. Since *minimize* and *maximize* are entirely symmetrical we shall discuss *maximize* and assume that *minimize* is treated similarly.) Once decomposed in this way, *maximize* can use $minimize'$, rather than *minimize* itself, to discover which numbers *minimize* 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 *maximize* 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's easy to "factor out" *max* from the definition of *maximize*, giving

$$\begin{aligned}
maximize'\ (Node\ n\ Nil) &= Cons\ n\ Nil \\
maximize'\ (Node\ n\ l\ ) &= map\ minimize\ l \\
&= map\ (min\ .\ minimize')\ l \\
&= map\ min\ (map\ minimize'\ l\ ) \\
&= mapmin\ (map\ minimize'\ l\ ) \\
&\quad \textbf{where}\ mapmin = map\ min
\end{aligned}$$

Since $minimize'$ returns a list of numbers, the minimum of which is the result of *minimize*, $(map\ minimize'\ l)$ returns a list of lists of numbers, and $maximize'$ should return a list of those lists' minima. Only the maximum of this list matters, however. We shall define a new version of *mapmin* that omits the minima of lists whose minimum doesn't matter.

$$\begin{aligned}
&mapmin\ (Cons\ nums\ rest) \\
&\quad = Cons\ (min\ nums)\ (omit\ (min\ nums)\ rest)
\end{aligned}$$

The function *omit* is passed a "potential maximum" — the largest minimum seen so far — and omits any minima that are less than this:

$$
\begin{aligned}
&omit\ pot\ Nil = Nil \\
&omit\ pot\ (Cons\ nums\ rest) \\
&\quad = omit\ pot\ rest, && \textbf{if}\ minleq\ nums\ pot \\
&\quad = Cons\ (min\ nums)\ (omit\ (min\ nums)\ rest), && \textbf{otherwise}
\end{aligned}
$$

The function *minleq* takes a list of numbers and a potential maximum, and it returns $True$ if the minimum of the list of numbers does not exceed the potential maximum. To do this, it does not need to look at the entire 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

$$
\begin{aligned}
&minleq\ Nil\ pot = False \\
&minleq\ (Cons\ n\ rest)\ pot = True, && \textbf{if}\ n \leq pot \\
&\qquad\qquad\qquad\qquad = minleq\ rest\ pot, && \textbf{otherwise}
\end{aligned}
$$

Having defined $maximize'$ and $minimize'$ in this way it is simple to write a new evaluator:

$$evaluate = max\ .\ maximize'\ .\ maptree\ static\ .\ prune\ 8\ .\ gametree$$

Thanks to lazy evaluation, the fact that $maximize'$ 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 optimizations in $maximize'$, although fairly simple, can have a dramatic effect on the speed of evaluation and so can allow the evaluator to look further ahead.

Other optimizations 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 subtrees 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

$$
\begin{aligned}
&highfirst\ (Node\ n\ sub) = Node\ n\ (sort\ higher\ (map\ lowfirst\ sub)) \\
&lowfirst\ (Node\ n\ sub) = Node\ n\ (sort\ (not\ .\ higher)\ (map\ highfirst\ sub)) \\
&higher\ (Node\ n1\ sub1)\ (Node\ n2\ sub2) = n1 > n2
\end{aligned}
$$

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

$$
\begin{aligned}
&evaluate \\
&\quad = max\ .\ maximize'\ .\ highfirst\ .\ maptree\ static\ .\ prune\ 8\ .\ gametree
\end{aligned}
$$

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 necessary only to replace *highfirst* with (*taketree* 3 . *highfirst*), where

$$\begin{aligned} &\mathit{taketree}\ n = \ \mathit{foldtree}\ (\mathit{nodett}\ n)\ \mathit{Cons}\ \mathit{Nil} \\ &\mathit{nodett}\ n\ \mathit{label}\ \mathit{sub} = \mathit{Node}\ \mathit{label}\ (\mathit{take}\ n\ \mathit{sub}) \end{aligned}$$

The function *taketree* replaces all the nodes in a tree with nodes that have 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's usual to define certain "dynamic" positions and not to allow look-ahead to stop in one of these. Assuming a function *dynamic* that recognizes such positions, we need only add one equation to *prune* to do this:

$$\begin{aligned} &\mathit{prune}\ 0\ (\mathit{Node}\ \mathit{pos}\ \mathit{sub}) \\ &\quad = \mathit{Node}\ \mathit{pos}\ (\mathit{map}\ (\mathit{prune}\ 0)\ \mathit{sub}), \quad \textbf{if}\ \mathit{dynamic}\ \mathit{pos} \end{aligned}$$

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 *maximize*, the last function in the chain, and *gametree*, the first, in the absence of lazy evaluation it could be written only as a monolithic program. Such a programs are 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 that 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 support 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 modularize programs in new and useful ways, and we've shown several examples of this. Smaller and more general modules can be reused 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 modularize it and to generalize the parts. He or she should expect to use higher-order functions and lazy evaluation as the 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 [3]. Abelson and Sussman stress that streams (lazy lists) are a powerful tool for structuring programs [1]. Henderson has used streams to structure functional operating systems [2]. But perhaps we place more stress on functional programs' modularity than previous authors.

This paper 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 [1]). 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.

## Acknowledgments

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, Göteborg, pointed out a serious error in an earlier version of one of the numerical algorithms, and thereby prompted development of many of the others. Ham Richards and David Turner did a superb editorial job, including converting the notation to Miranda. This work was carried out with the support of a Research Fellowship from the U.K. Science and Engineering Research Council.

## References

[1] Abelson, H. and Sussman, G. J. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1984.

[2] Henderson, P. "Purely functional operating systems". In *Functional Programming and its Applications*. Cambridge University Press, Cambridge, 1982.

[3] Turner, D. A. "The semantic elegance of applicative languages". In *ACM Symposium on Functional Languages and Computer Architecture* (Wentworth, N.H.). ACM, New York, 1981.

[4] Turner, D. A. "An Overview of Miranda". *SIGPLAN Notices*, December 1986 (this and other papers about Miranda are at: http://miranda.org.uk).

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

[6] Wirth, N. *Programming in Modula–II*. Springer-Verlag, Berlin, 1982.