Functional Programming
Paper Collection CPSC 521

September 12, 2011

1 Table of Contents

Why Functional Programming Matters

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

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!*

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 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 **goto**s" 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 **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 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

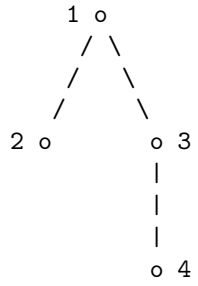
```
        summatrix = sum . map sum
```

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



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 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.

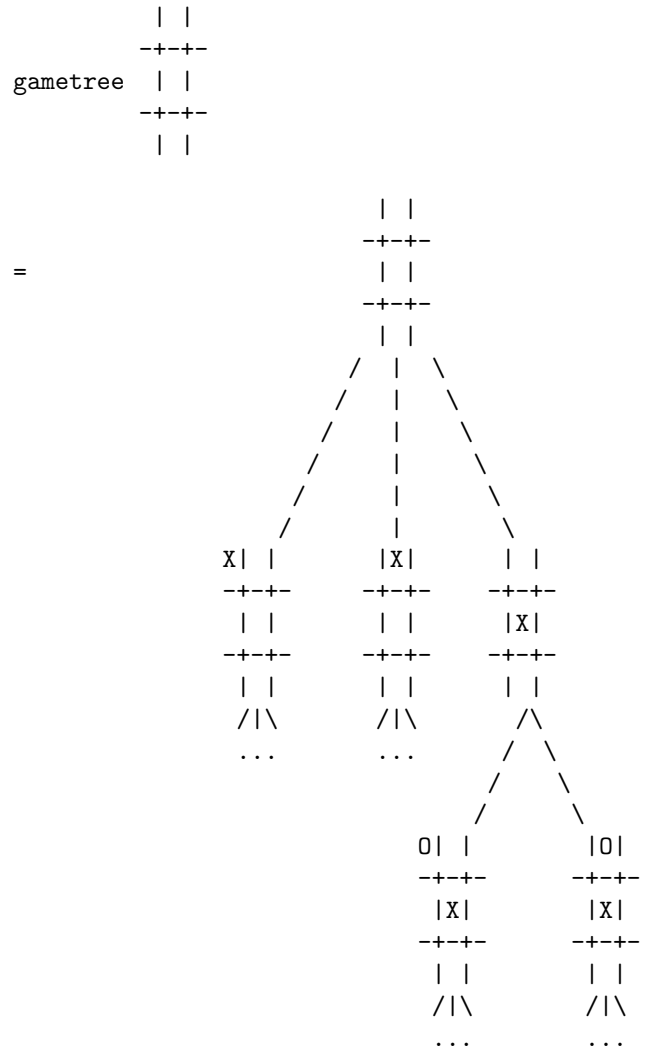


Figure 1: An Example of a Game-Tree.

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

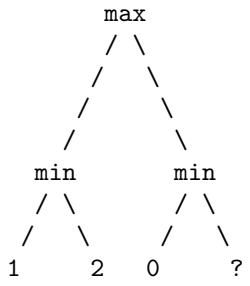
```
prune 5 . gametree
```

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:



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.

Where do I begin? A problem solving approach in teaching functional programming

Simon Thompson

Computing Laboratory
University of Kent at Canterbury
S.J.Thompson@ukc.ac.uk

Abstract. This paper introduces a problem solving method for teaching functional programming, based on Polya's *How To Solve It*, an introductory investigation of mathematical method. We first present the language independent version, and then show in particular how it applies to the development of programs in Haskell. The method is illustrated by a sequence of examples and a larger case study.

Keywords. Functional programming, Haskell, palindrome recognition, Polya, problem solving.

1 Introduction

Many students take easily to functional programming, whilst others experience difficulties of one sort or another. The work reported here is the result of attempts to advise students on how to use problem solving ideas to help them design and develop programs.

Some students come to a computer science degree with considerable experience of programming in an imperative language such as Pascal or C. For these students, a functional approach forces them to look afresh at the process of programming; it is no longer possible to construct programs 'from the middle out'; instead design has to be confronted from the start. Other students come to a CS programme with no prior programming experience, and so with no 'baggage' which might encumber them. Many of these students prefer a functional approach to the imperative, but lacking the background of the experienced students need encouragement and advice about how to build programs.[1]

In this paper we report on how we try to answer our students' question '*Where do I begin?*' by talking explicitly about problem solving and what it means in programming. Beyond enabling students to program more effectively a problem solving approach has a number of other important consequences. The approach is not only beneficial in a functional programming context, as we are able to use the approach across our introductory curriculum, as reported in [1], reinforcing ideas in a disparate set of courses including imperative programming and systems analysis. It is also striking that the cycle of problem solving is very

[1] Further reports on instructors' experience of teaching functional programming were given at the recent Workshop in the UK [6].

close to the 'understand, plan, write and review' scheme which is recommended to students experiencing difficulties in writing essays, emphasising the fact that problem solving ability is a transferable skill.

In this paper we first review our general problem solving strategy, modelled on Polya's epoch-making *How To Solve It*, [5], which brought these ideas to prominence in mathematics some fifty years ago. This material is largely language-independent. We then go on to explore how to take these ideas into the functional domain by describing '*How to program it in Haskell*'. After looking at a sequence of examples we examine the case study of palindrome recognition, and the lessons to be learned from this example. We conclude by reviewing the future role of problem solving in functional programming and across the computer science curriculum, since the material on problem solving can also be seen as the first stage in learning software engineering, 'in the small' as it were; more details are given in [1].

I am very grateful to David Barnes and Sally Fincher with whom the cross-curricular ideas were developed, and to Jan Sellers of the Rutherford Study Centre at the University of Kent who provided support for workshops in problem solving, as well as pointing out the overlap with essay writing techniques. The Alumni Fund of the University of Kent provided funding for Jan to work with us. Finally I would like to acknowledge all the colleagues at UKC with whom I have taught functional programming, and from whom I have learned an immense amount.

2 How To Program It

Polya's *How To Solve It*, [5], contains a wealth of material about how to approach mathematical problems of various kinds. This ranges from specific hints which can be used in particular circumstances to general methodological ideas. The latter are summarised in a two-page table giving a four-stage process (or more strictly a cycle) for solving problems. In helping students to program, we have specified a similar summary of method – *How To Program It* – which is presented in Figures 1 and 2. The stages of our cycle are: understanding the problem; designing the program; writing the program and finally looking back (or 'reflection').

The table is largely self-explanatory, so we will not paraphrase it here; instead we will make some comments about its structure and how it has been used.

How To Program It has been written in a language-independent way (at least as much as the terminology of modern computing allows). In Section 3 we look at how it can be specialised for the lazy functional programming language Haskell, [4, 7]. Plainly it can also be used with other programming languages, and at the University of Kent we have used it in teaching Modula-3, [1], for instance.

Our approach emphasizes that a novice can make substantial progress in completing a programming task *before* beginning to write any program code. This is very important in demystifying the programming process for those who find it difficult. As the title of this paper suggests, getting started in the task

UNDERSTANDING THE PROBLEM

First understand the problem.

What are the inputs (or arguments)? What are the outputs (or results)? What is the specification of the problem?

Name the program or function.

What is its type?

Can the specification be satisfied? Is it insufficient? or redundant? or contradictory? What special conditions are there on the inputs and outputs?

Does the problem break into parts? It can help to draw diagrams and to write things down in pseudo-code or plain English.

DESIGNING THE PROGRAM

In designing the program you need to think about the connections between the input and the output.

If there is no immediate connection, you might have to think of auxiliary problems which would help in the solution.

You want to give yourself some sort of plan of how to write the program.

Have you seen the problem before? In a slightly different form?

Do you know a related problem? Do you know any programs or functions which could be useful?

Look at the specification. Try to find a familiar problem with the same or similar specification.

Here is a problem related to yours and solved before. Could you use it? Could you use its results? Could you use its methods? Should you introduce some auxiliary parts to the program?

If you cannot solve the proposed problem try to solve a related one. Can you imagine a more accessible related one? A more general one? A more specific one? An analogous problem?

Can you solve part of the problem? Can you get something useful from the inputs? Can you think of information which would help you to calculate the outputs? How could you change the inputs/outputs so that they were *closer* to each other?

Did you use all the inputs? Did you use the special conditions on the inputs? Have you taken into account all that the specification requires?

Fig. 1. How To Program It, Part I

WRITING YOUR PROGRAM

Writing the program means taking your design into a particular programming language.

In writing your program, make sure that you check each step of the design. Can you see clearly that each step does what it should?

Think about how you can build programs in the language. How do you deal with different cases? With doing things in sequence? With doing things repeatedly or recursively?

You can write the program in stages. Think about the different cases into which the problem divides; in particular think about the different cases for the inputs. You can also think about computing parts of the result separately, and how to put the parts together to get the final results.

You can think of solving the problem by solving it for a smaller input and using the result to get your result; this is recursion.

You also need to know the programs you have already written, and the functions built into the language or library.

Your design may call on you to solve a more general or more specific problem. Write the solutions to these; they may guide how you write the solution itself, or may indeed be used in that solution.

You should also draw on other programs you have written. Can they be used? Can they be modified? Can they guide how to build the solution?

LOOKING BACK

Examine your solution: how can it be improved?

Can you test that the program works, on a variety of arguments?

Can you think of how you might write the program differently if you had to start again?

Can you see how you might use the program or its method to build another program?

Fig. 2. How To Program It, Part II

can be a block for many students. For example, in the first stage of the process a student will have to clarify the problem in two complementary ways. First, the informal statement has to be clarified, and perhaps restated, giving a clear informal goal. Secondly, this should mean that the student is able to write down the name of a program or function and more importantly give a type to this artifact at this stage. While this may seem a small step, it means that misconceptions can be spotted at an early stage, and avoid a student going off in a mistaken direction.

The last observation is an example of a general point. Although we have made

reflection (or 'looking back') the final stage of the process, it should permeate the whole process. At the first stage, once a type for a function has been given, it is sensible to reflect on this choice: giving some typical inputs and corresponding outputs, does the type specified actually reflect the problem? This means that a student is forced to check both their understanding of the problem and of the types of the target language.

At the design stage, students are encouraged to think about the context of the problem, and the ways in which this can help the solution of the problem itself. We emphasise that programs can be re-used either by calling them or by modifying their definitions, as well as the ideas of specialisation and generalisation. Generalisation is particularly apt in the modern functional context, in which polymorphism and higher-order functions allow libraries of general functions to be written with little overhead (in contrast to the C++ Standard Template Library, say).

Implementation ideas can be discussed in a more concrete way in the context of a particular language. The ideas of this section are next discussed in the context of Haskell by means of a succession of examples in Section 3 and by a lengthier case study in Section 4. Note that the design stage of the case study is essentially language independent.

Students are encouraged to reflect on what they have achieved throughout the problem solving cycle. As well as testing their finished programs, pencil and paper evaluation of Haskell programs is particularly effective, and we expect students to use this as a way of discovering how their programs work.

3 Programming it in Haskell

As we saw in the previous section, it is more difficult to give useful language-independent advice about how to write programs than it is about how to design them. It is also easier to understand the generalities of *How To Program It* in the context of particular examples. We therefore provide students with particular language-specific advice in tabular form. These tables allow us to

- give examples to illustrate the design and programming stages of the process, and
- discuss the programming process in a much more specific way.

The full text of *Programming it in Haskell* is available on the World Wide Web, [9]. Rather than reproduce it here, in the rest of this section we look at some of the examples and the points in the process which they illustrate.

Problem: find the maximum of three integers

A first example is to find the maximum of three integers. In our discussion we link various points in the exposition to the four stages of *How To Program It*.

Understanding the problem Even in a problem of this simplicity there can be some discussion of the specification: what is to be done in the case when two (or three) of the integers are maximal? This is usually resolved by saying that the common value should be returned, but the important learning point here is that the discussion takes place. Also one can state the name and type, beginning the solution:

```
maxThree :: Int -> Int -> Int -> Int
```

Designing and writing the program More interesting points can be made in the design stage. Given a function `max` to find the maximum of two integers,

```
max :: Int -> Int -> Int
max a b
  | a>=b         = a
  | otherwise    = b
```

this can be used in two ways. It can form a model for the solution of the problem:

```
maxThree a b c
  | a>=b && a>=c     = a
  |    ....
```

or it can itself be used in a solution

```
maxThree a b c = max (max a b) c
```

It is almost universally the case that novices produce the first solution rather than the second, so this provides a useful first lesson in the existence of design choices, guided by the resources available (in this case the function `max`). Although it is difficult to interpret exactly why this is the case, it can be taken as an indication that novice students find it more natural to tackle a problem in a single step, rather than stepping back from the problem and looking at it more strategically. This lends support to introducing these problem solving ideas explicitly, rather than hoping that they will be absorbed 'osmotically'.

We also point out that given `maxThree` it is straightforward to generalise to cases of finding the minimum of three numbers, the maximum of four, and so on.

Looking back Finally, this is a non-trivial example for program testing. A not uncommon student error here is to make the inequalities strict, thus

```
maxThreeErr a b c
  | a>b && a>c     = a
  | b>c && b>a     = b
  | otherwise      = c
```

This provides a discussion point in how test data are chosen; the vast majority of student test data sets do not reveal the error. A systematic approach should produce the data which indicate the error – `a` and `b` jointly maximal – and indeed the cause of error links back to the initial clarification of the specification.

Problem: add the positive numbers in a list

We use this example to show how to break down the process of *designing* and *writing* a program – stages two and three of our four-step process – into a number of simpler steps. The function we require is

```
addPos :: [Int] -> Int
```

We first consider the design of the equations which describe the function. A paradigm here if we are to define the function from scratch is *primitive recursion* (or structural recursion) over the list argument. In doing this we adopt the general scheme

```
addPos []    = ...
addPos (a:x) = ... addPos x ...
```

in which we have do define the value at `[]` outright and the value at `(a:x)` from the value at `x`. Completing the first equation gives

```
addPos []    = 0
```

The `(a:x)` case requires more thought. Guidance can often come from looking at examples. Here we take lists

```
[-4,3,2,-1]
[2,3,2,-1]
```

which respectively give sums `0` and `6`. In the first case the head does not contribute to the sum; in the second it does. This suggests the case analysis

```
addPos (a:x)
  | a>0         = ...
  | otherwise   = ...
```

from which point in development the answer can be seen. The point of this example is less to develop the particular function than to illustrate how the process works.

The example is also enlightening for the other design possibilities it offers by way of *looking back* at the problem. In particularly when students are acquainted with `filter` and `foldr` the explicit definition

```
addPos = foldr (+) 0 . filter (>0)
```

is possible. The definition here reflects very clearly its top-down design.

Further examples

Other examples we have used include

Maximum of a list This is similar to `addPos`, but revisits the questions raised by the `maxThree` example. In particular, will the `max` function be used in the definition?

Counting how many times a maximum occurs among three numbers This gives a reasonable example in which local definitions (in a `where` clause) naturally structure a definition with a number of parts.

Deciding whether one list is a sublist of another This example naturally gives rise to an auxiliary function during its development.

Summing integers up to `n` This can give rise to the generalisation of summing numbers from `m` to `n`.

The discussions thus far have been about algorithms; there is a wealth of material which addresses data and object design, the former of which we address in [9].

4 Case study: palindromes

The problem is to recognise palindromes, such as

```
"Madam I'm Adam"
```

It is chosen as an example since even for a more confident student it requires some thought before implementation can begin. Once the specification is clarified it presents a non-trivial design space in which we can illustrate how choices between alternative designs can take place. Indeed, it is a useful example for small-group work since it is likely that different groups will produce substantially different initial design ideas. It is also an example in which a variety of standard functions can be used.

We address the main ideas in this section; further details are available on the World Wide Web [8].

Understanding the problem

The problem is stated in a deliberately vague way. A palindrome can be identified as a string which is the same read forwards or backwards, so long as

(1) we disregard the punctuation (punctuation marks and spaces) in the string;
(2) we disregard the case (upper or lower: that is capital or small) of the letters in the string.

Requirement (2) is plainly unambiguous, whilst (1) will need to be revisited at the implementation stage.

Overall design

The palindrome example lends itself to a wide choice of designs. The **simpler problem** in which there is no punctuation and all letters in lower case can be helpful in two ways. It can either form a *guide* about how to write the full solution, or be *used* as a part of that solution. The choice here provides a useful discussion point.

Design: the simpler problem

Possible designs which can emerge here may be classified in two different ways.

- Is the string handled as a single entity, or split into two parts?
- Is comparison made between strings, or between individual characters?

These choices generate these outline designs:

- The string is reversed and compared with itself;
- the string is split, one part reversed and the result compared with the other part;
- the first and last characters are compared, and if equal are removed and an iteration or a recursion is performed;
- the string is split, one part reversed and the strings are then compared one character at a time.

Again, it is important for students to be able both to see the possibilities available, and to discuss their relative merits (in the context of the implementation language). Naturally, too, there needs to be a comparison of the different ways in which the string is represented.

Design: the full problem

Assuming we are to use the solution to the simpler problem in solving the full problem, we reach our goal by writing a function which removes punctuation and changes all upper case letters to lower case. Here again we can see an opportunity to split the task in two, and also to discuss the order in which the two operations are performed: do we remove punctuation before or after converting letters to lower case? This allows a discussion of relative efficiency.

Writing the program

At this point we need to revisit the specification and to make plain what is meant by punctuation. This is not clear from the example given in the specification, and we can choose either to be proscriptive and disallow everything but letters and digits, or to be permissive and to say that punctuation consists of a particular set of characters.

There are more specific implementation decisions to be taken here; these reinforce the discussions in Section 3. In particular there is substantial scope for using built-in or library functions.

We give a full implementation of the palindrome recognition problem in Figure 3.

```
palin :: String -> Bool

palin st = simplePalin (disregard st)

simplePalin :: String -> Bool

simplePalin st = (rev st == st)

rev :: String -> String

rev []    = []
rev (a:st) = rev st ++ [a]

disregard :: String -> String

disregard st = change (remove st)

remove :: String -> String
change :: String -> String

remove [] = []
remove (a:st)
  | notPunct a  = a : remove st
  | otherwise   =     remove st

notPunct ch = isAlpha ch || isDigit ch

change [] = []
change (a:st) = convert a : change st

convert :: Char -> Char

convert ch
  | isCap ch      = toEnum (fromEnum ch + offset)
  | otherwise     = ch
    where
    offset = fromEnum 'a' - fromEnum 'A'

isCap :: Char -> Bool

isCap ch = 'A' <= ch && ch <= 'Z'
```

Fig. 3. Recognising palindromes in Haskell

Looking back

Using the approach suggested here, students see that the solution which they have chosen represents one branch in a tree of choices. Their solution can be evaluated against other possibilities, including those written by other students. There is also ample scope for discussion of testing in this problem.

For instance, the solution given in Figure 3 can give rise to numerous discussion points.

- No higher order functions are used in the solution; we would expect to revisit the example after covering HOFs to reveal that `change` is `map convert` and that `remove` is `filter notPunct`.
- In a similar way we would expect to revisit the solution and discuss incorporating function-level definitions such as

  ```
  palin = simplePalin . disregard
  ```

  This would also apply to `disregard` itself.
- Some library functions have been used; digits and letters are recognised by `isDigit` and `isAlpha`.
- An alternative definition of `disregard` is given by

  ```
  disregard st = remove (change st)
  ```

  and other solutions are provided by implementing the two operations in a single function definition, rather than as a composition of two separate pieces of functionality.
- We have chosen the proscriptive definition of punctuation, considering only letters and digits to be significant.

5 Conclusion

In this paper we have given an explicit problem solving method for beginning (functional) programmers, motivated by the desire to equip them with tools to enable them to write complex programs in a disciplined way. The method also gives weaker students the confidence to proceed by showing them the ways in which a seemingly intractable problem can be broken down into simpler parts which can be solved separately. As well as providing a general method we think it crucial to illustrate the method by examples and case studies – this latter approach is not new, see [2] for a very effective account of using case studies in teaching Pascal.

To conclude, it is worth noting that numerous investigations into mathematical method were stimulated by Polya's work. Most prominent are Lakatos' investigations of the roles of proof and counterexample, [3], which we believe have useful parallels for teachers and students of computer science. We intend to develop this correspondence further in the future.

References

1. David Barnes, Sally Fincher, and Simon Thompson. Introductory problem solving in computer science. In *CTC97, Dublin*, 1997.
2. Michael Clancy and Marcia Linn. *Designing Pascal Solutions: Case studies using data structures*. Computer Science Press, W. H. Freeman and Co., 1996.
3. Imre Lakatos. *Proofs and Refuations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976. Edited by John Worrall and Elie Zahar.
4. John Peterson and Kevin Hammond, editors. *Report on the Programming Language Haskell, Version 1.3*. `http://haskell.cs.yale.edu/haskell-report/haskell-report.html`, 1996.
5. G. Polya. *How To Solve It*. Princeton University Press, second edition, 1957.
6. Teaching functional programming: Opportunities & difficulties. `http://www.ukc.ac.uk/CSDN/conference/96/Report.html`, September 1996.
7. Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.
8. Simon Thompson. Problem solving: recognising palindromes. `http://www.ukc.ac.uk/computer_science/Haskell_craft/palindrome.html`, 1996.
9. Simon Thompson. Programming it in Haskell. `http://www.ukc.ac.uk/computer_science/Haskell_craft/ProgInHaskell.html`, 1996.

This article was processed using the LaTeX macro package with LLNCS style

A Gentle Introduction to Haskell 98

Paul Hudak
Yale University
Department of Computer Science

John Peterson
Yale University
Department of Computer Science

Joseph H. Fasel
University of California
Los Alamos National Laboratory

October, 1999

1 Introduction

Our purpose in writing this tutorial is not to teach programming, nor even to teach functional programming. Rather, it is intended to serve as a supplement to the Haskell Report [4], which is otherwise a rather dense technical exposition. Our goal is to provide a gentle introduction to Haskell for someone who has experience with at least one other language, preferably a functional language (even if only an "almost-functional" language such as ML or Scheme). If the reader wishes to learn more about the functional programming style, we highly recommend Bird's text *Introduction to Functional Programming* [1] or Davie's *An Introduction to Functional Programming Systems Using Haskell* [2]. For a useful survey of functional programming languages and techniques, including some of the language design principles used in Haskell, see [3].

The Haskell language has evolved significantly since its birth in 1987. This tutorial deals with Haskell 98. Older versions of the language are now obsolete; Haskell users are encouraged to use Haskell 98. There are also many extensions to Haskell 98 that have been widely implemented. These are not yet a formal part of the Haskell language and are not covered in this tutorial.

Our general strategy for introducing language features is this: motivate the idea, define some terms, give some examples, and then point to the Report for details. We suggest, however, that the reader completely ignore the details until the *Gentle Introduction* has been completely read. On the

other hand, Haskell's Standard Prelude (in Appendix A of the Report and the standard libraries (found in the Library Report [5]) contain lots of useful examples of Haskell code; we encourage a thorough reading once this tutorial is completed. This will not only give the reader a feel for what real Haskell code looks like, but will also familiarize her with Haskell's standard set of predefined functions and types.

Finally, the Haskell web site, `http://haskell.org`, has a wealth of information about the Haskell language and its implementations.

[We have also taken the course of not laying out a plethora of lexical syntax rules at the outset. Rather, we introduce them incrementally as our examples demand, and enclose them in brackets, as with this paragraph. This is in stark contrast to the organization of the Report, although the Report remains the authoritative source for details (references such as "§2.1" refer to sections in the Report).]

Haskell is a *typeful* programming language:[1] types are pervasive, and the newcomer is best off becoming well aware of the full power and complexity of Haskell's type system from the outset. For those whose only experience is with relatively "untypeful" languages such as Perl, Tcl, or Scheme, this may be a difficult adjustment; for those familiar with Java, C, Modula, or even ML, the adjustment should be easier but still not insignificant, since Haskell's type system is different and somewhat richer than most. In any case, "typeful programming" is part of the Haskell programming experience, and cannot be avoided.

2 Values, Types, and Other Goodies

Because Haskell is a purely functional language, all computations are done via the evaluation of *expressions* (syntactic terms) to yield *values* (abstract entities that we regard as answers). Every value has an associated *type*. (Intuitively, we can think of types as sets of values.) Examples of expressions include atomic values such as the integer `5`, the character `'a'`, and the function `\x -> x+1`, as well as structured values such as the list `[1,2,3]` and the pair `('b',4)`.

Just as expressions denote values, type expressions are syntactic terms that denote type values (or just *types*). Examples of type expressions include the atomic types `Integer` (infinite-precision integers), `Char` (characters), `Integer->Integer` (functions mapping `Integer` to `Integer`), as well as the structured types `[Integer]` (homogeneous lists of integers) and `(Char,Integer)` (character, integer pairs).

All Haskell values are "first-class"—they may be passed as arguments to functions, returned as results, placed in data structures, etc. Haskell types, on the other hand, are *not* first-class. Types in a sense describe values, and the association of a value with its type is called a *typing*. Using the examples of values and types above, we write typings as follows:

```
  5  :: Integer
'a'  :: Char
inc  :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

[1] Coined by Luca Cardelli.

The "::" can be read "has type."

Functions in Haskell are normally defined by a series of *equations*. For example, the function `inc` can be defined by the single equation:

```
inc n           = n+1
```

An equation is an example of a *declaration*. Another kind of declaration is a *type signature declaration* (§4.4.1), with which we can declare an explicit typing for `inc`:

```
inc             :: Integer -> Integer
```

We will have much more to say about function definitions in Section 3.

For pedagogical purposes, when we wish to indicate that an expression e_1 evaluates, or "reduces," to another expression or value e_2, we will write:

$$e_1 \quad \Rightarrow \quad e_2$$

For example, note that:

```
inc (inc 3)    ⇒    5
```

Haskell's static type system defines the formal relationship between types and values (§4.1.3). The static type system ensures that Haskell programs are *type safe*; that is, that the programmer has not mismatched types in some way. For example, we cannot generally add together two characters, so the expression `'a'+'b'` is ill-typed. The main advantage of statically typed languages is well-known: All type errors are detected at compile-time. Not all errors are caught by the type system; an expression such as `1/0` is typable but its evaluation will result in an error at execution time. Still, the type system finds many program errors at compile time, aids the user in reasoning about programs, and also permits a compiler to generate more efficient code (for example, no run-time type tags or tests are required).

The type system also ensures that user-supplied type signatures are correct. In fact, Haskell's type system is powerful enough to allow us to avoid writing any type signatures at all;[2] we say that the type system *infers* the correct types for us. Nevertheless, judicious placement of type signatures such as that we gave for `inc` is a good idea, since type signatures are a very effective form of documentation and help bring programming errors to light.

[The reader will note that we have capitalized identifiers that denote specific types, such as `Integer` and `Char`, but not identifiers that denote values, such as `inc`. This is not just a convention: it is enforced by Haskell's lexical syntax. In fact, the case of the other characters matters, too: `foo`, `fOo`, and `fOO` are all distinct identifiers.]

2.1 Polymorphic Types

Haskell also incorporates *polymorphic* types—types that are universally quantified in some way over all types. Polymorphic type expressions essentially describe families of types. For example, (∀`a`)`[a]` is the family of types consisting of, for every type `a`, the type of lists of `a`. Lists of

[2] With a few exceptions to be described later.

integers (e.g. `[1,2,3]`), lists of characters (`['a','b','c']`), even lists of lists of integers, etc., are all members of this family. (Note, however, that `[2,'b']` is *not* a valid example, since there is no single type that contains both `2` and `'b'`.)

[Identifiers such as `a` above are called *type variables*, and are uncapitalized to distinguish them from specific types such as `Int`. Furthermore, since Haskell has only universally quantified types, there is no need to explicitly write out the symbol for universal quantification, and thus we simply write `[a]` in the example above. In other words, all type variables are implicitly universally quantified.]

Lists are a commonly used data structure in functional languages, and are a good vehicle for explaining the principles of polymorphism. The list `[1,2,3]` in Haskell is actually shorthand for the list `1:(2:(3:[]))`, where `[]` is the empty list and `:` is the infix operator that adds its first argument to the front of its second argument (a list).[3] Since `:` is right associative, we can also write this list as `1:2:3:[]`.

As an example of a user-defined function that operates on lists, consider the problem of counting the number of elements in a list:

```
length                  :: [a] -> Integer
length []               =  0
length (x:xs)           =  1 + length xs
```

This definition is almost self-explanatory. We can read the equations as saying: "The length of the empty list is 0, and the length of a list whose first element is `x` and remainder is `xs` is 1 plus the length of `xs`." (Note the naming convention used here; `xs` is the plural of `x`, and should be read that way.)

Although intuitive, this example highlights an important aspect of Haskell that is yet to be explained: *pattern matching*. The left-hand sides of the equations contain patterns such as `[]` and `x:xs`. In a function application these patterns are matched against actual parameters in a fairly intuitive way (`[]` only matches the empty list, and `x:xs` will successfully match any list with at least one element, binding `x` to the first element and `xs` to the rest of the list). If the match succeeds, the right-hand side is evaluated and returned as the result of the application. If it fails, the next equation is tried, and if all equations fail, an error results.

Defining functions by pattern matching is quite common in Haskell, and the user should become familiar with the various kinds of patterns that are allowed; we will return to this issue in Section 4.

The `length` function is also an example of a polymorphic function. It can be applied to a list containing elements of any type, for example `[Integer]`, `[Char]`, or `[[Integer]]`.

```
length [1,2,3]          ⇒    3
length ['a','b','c']    ⇒    3
length [[1],[2],[3]]    ⇒    3
```

Here are two other useful polymorphic functions on lists that will be used later. Function `head` returns the first element of a list, function `tail` returns all but the first.

[3] `:` and `[]` are like Lisp's `cons` and `nil`, respectively.

```
head                    :: [a] -> a
head (x:xs)             =  x

tail                    :: [a] -> [a]
tail (x:xs)             =  xs
```

Unlike `length`, these functions are not defined for all possible values of their argument. A runtime error occurs when these functions are applied to an empty list.

With polymorphic types, we find that some types are in a sense strictly more general than others in the sense that the set of values they define is larger. For example, the type `[a]` is more general than `[Char]`. In other words, the latter type can be derived from the former by a suitable substitution for `a`. With regard to this generalization ordering, Haskell's type system possesses two important properties: First, every well-typed expression is guaranteed to have a unique principal type (explained below), and second, the principal type can be inferred automatically (§4.1.3). In comparison to a monomorphically typed language such as C, the reader will find that polymorphism improves expressiveness, and type inference lessens the burden of types on the programmer.

An expression's or function's principal type is the least general type that, intuitively, "contains all instances of the expression". For example, the principal type of `head` is `[a]->a`; `[b]->a`, `a->a`, or even `a` are correct types, but too general, whereas something like `[Integer]->Integer` is too specific. The existence of unique principal types is the hallmark feature of the *Hindley-Milner type system*, which forms the basis of the type systems of Haskell, ML, Miranda,[4] and several other (mostly functional) languages.

2.2 User-Defined Types

We can define our own types in Haskell using a `data` declaration, which we introduce via a series of examples (§4.2.1).

An important predefined type in Haskell is that of truth values:

```
data Bool               = False | True
```

The type being defined here is `Bool`, and it has exactly two values: `True` and `False`. Type `Bool` is an example of a (nullary) *type constructor*, and `True` and `False` are (also nullary) *data constructors* (or just *constructors*, for short).

Similarly, we might wish to define a color type:

```
data Color              = Red | Green | Blue | Indigo | Violet
```

Both `Bool` and `Color` are examples of enumerated types, since they consist of a finite number of nullary data constructors.

Here is an example of a type with just one data constructor:

```
data Point a            = Pt a a
```

Because of the single constructor, a type like `Point` is often called a *tuple type*, since it is essentially

[4] "Miranda" is a trademark of Research Software, Ltd.

just a cartesian product (in this case binary) of other types.[5] In contrast, multi-constructor types, such as `Bool` and `Color`, are called (disjoint) union or sum types.

More importantly, however, `Point` is an example of a polymorphic type: for any type t, it defines the type of cartesian points that use t as the coordinate type. The `Point` type can now be seen clearly as a unary type constructor, since from the type t it constructs a new type `Point` t. (In the same sense, using the list example given earlier, `[]` is also a type constructor. Given any type t we can "apply" `[]` to yield a new type `[`t`]`. The Haskell syntax allows `[]` t to be written as `[`t`]`. Similarly, `->` is a type constructor: given two types t and u, t`->`u is the type of functions mapping elements of type t to elements of type u.)

Note that the type of the binary data constructor `Pt` is `a -> a -> Point a`, and thus the following typings are valid:

```
Pt  2.0  3.0          :: Point Float
Pt  'a'  'b'          :: Point Char
Pt True False         :: Point Bool
```

On the other hand, an expression such as `Pt 'a' 1` is ill-typed because `'a'` and `1` are of different types.

It is important to distinguish between applying a *data constructor* to yield a *value*, and applying a *type constructor* to yield a *type*; the former happens at run-time and is how we compute things in Haskell, whereas the latter happens at compile-time and is part of the type system's process of ensuring type safety.

[Type constructors such as `Point` and data constructors such as `Pt` are in separate namespaces. This allows the same name to be used for both a type constructor and data constructor, as in the following:

```
data Point a = Point a a
```

While this may seem a little confusing at first, it serves to make the link between a type and its data constructor more obvious.]

2.2.1 Recursive Types

Types can also be recursive, as in the type of binary trees:

```
data Tree a             = Leaf a | Branch (Tree a) (Tree a)
```

Here we have defined a polymorphic binary tree type whose elements are either leaf nodes containing a value of type `a`, or internal nodes ("branches") containing (recursively) two sub-trees.

When reading data declarations such as this, remember again that `Tree` is a type constructor, whereas `Branch` and `Leaf` are data constructors. Aside from establishing a connection between these constructors, the above declaration is essentially defining the following types for `Branch` and `Leaf`:

```
Branch                  :: Tree a -> Tree a -> Tree a
Leaf                    :: a -> Tree a
```

[5] Tuples are somewhat like records in other languages.

With this example we have defined a type sufficiently rich to allow defining some interesting (recursive) functions that use it. For example, suppose we wish to define a function `fringe` that returns a list of all the elements in the leaves of a tree from left to right. It's usually helpful to write down the type of new functions first; in this case we see that the type should be `Tree a -> [a]`. That is, `fringe` is a polymorphic function that, for any type `a`, maps trees of `a` into lists of `a`. A suitable definition follows:

```
fringe                      :: Tree a -> [a]
fringe (Leaf x)             =  [x]
fringe (Branch left right) =  fringe left ++ fringe right
```

Here `++` is the infix operator that concatenates two lists (its full definition will be given in Section 9.1). As with the `length` example given earlier, the `fringe` function is defined using pattern matching, except that here we see patterns involving user-defined constructors: `Leaf` and `Branch`. [Note that the formal parameters are easily identified as the ones beginning with lower-case letters.]

2.3 Type Synonyms

For convenience, Haskell provides a way to define *type synonyms*; i.e. names for commonly used types. Type synonyms are created using a `type` declaration (§4.2.2). Here are several examples:

```
type String             = [Char]
type Person             = (Name,Address)
type Name               = String
data Address            = None | Addr String
```

Type synonyms do not define new types, but simply give new names for existing types. For example, the type `Person -> Name` is precisely equivalent to `(String,Address) -> String`. The new names are often shorter than the types they are synonymous with, but this is not the only purpose of type synonyms: they can also improve readability of programs by being more mnemonic; indeed, the above examples highlight this. We can even give new names to polymorphic types:

```
type AssocList a b              = [(a,b)]
```

This is the type of "association lists" which associate values of type `a` with those of type `b`.

2.4 Built-in Types Are Not Special

Earlier we introduced several "built-in" types such as lists, tuples, integers, and characters. We have also shown how new user-defined types can be defined. Aside from special syntax, are the built-in types in any way more special than the user-defined ones? The answer is *no*. The special syntax is for convenience and for consistency with historical convention, but has no semantic consequences.

We can emphasize this point by considering what the type declarations would look like for these built-in types if in fact we were allowed to use the special syntax in defining them. For example, the `Char` type might be written as:

```
data Char       = 'a' | 'b' | 'c' | ...         -- This is not valid
                | 'A' | 'B' | 'C' | ...         -- Haskell code!
                | '1' | '2' | '3' | ...
                ...
```

These constructor names are not syntactically valid; to fix them we would have to write something like:

```
data Char       = Ca | Cb | Cc | ...
                | CA | CB | CC | ...
                | C1 | C2 | C3 | ...
                ...
```

Even though these constructors are more concise, they are quite unconventional for representing characters.

In any case, writing "pseudo-Haskell" code in this way helps us to see through the special syntax. We see now that `Char` is just an enumerated type consisting of a large number of nullary constructors. Thinking of `Char` in this way makes it clear that we can pattern-match against characters in function definitions, just as we would expect to be able to do so for any of a type's constructors.

[This example also demonstrates the use of *comments* in Haskell; the characters `--` and all subsequent characters to the end of the line are ignored. Haskell also permits *nested* comments which have the form `{-...-}` and can appear anywhere (§2.2).]

Similarly, we could define `Int` (fixed precision integers) and `Integer` by:

```
data Int     = -65532 | ... | -1 | 0 | 1 | ... | 65532  -- more pseudo-code
data Integer =       ... -2 | -1 | 0 | 1 | 2 ...
```

where `-65532` and `65532`, say, are the maximum and minimum fixed precision integers for a given implementation. `Int` is a much larger enumeration than `Char`, but it's still finite! In contrast, the pseudo-code for `Integer` is intended to convey an *infinite* enumeration.

Tuples are also easy to define playing this game:

```
data (a,b)              = (a,b)                         -- more pseudo-code
data (a,b,c)            = (a,b,c)
data (a,b,c,d)          = (a,b,c,d)
  .                        .
  .                        .
  .                        .
```

Each declaration above defines a tuple type of a particular length, with `(...)` playing a role in both the expression syntax (as data constructor) and type-expression syntax (as type constructor). The vertical dots after the last declaration are intended to convey an infinite number of such declarations, reflecting the fact that tuples of all lengths are allowed in Haskell.

Lists are also easily handled, and more interestingly, they are recursive:

```
data [a]                = [] | a : [a]                  -- more pseudo-code
```

We can now see clearly what we described about lists earlier: `[]` is the empty list, and `:` is the infix

list constructor; thus `[1,2,3]` must be equivalent to the list `1:2:3:[]`. (`:` is right associative.) The type of `[]` is `[a]`, and the type of `:` is `a->[a]->[a]`.

[The way ":" is defined here is actually legal syntax—infix constructors are permitted in `data` declarations, and are distinguished from infix operators (for pattern-matching purposes) by the fact that they must begin with a ":" (a property trivially satisfied by ":").]

At this point the reader should note carefully the differences between tuples and lists, which the above definitions make abundantly clear. In particular, note the recursive nature of the list type whose elements are homogeneous and of arbitrary length, and the non-recursive nature of a (particular) tuple type whose elements are heterogeneous and of fixed length. The typing rules for tuples and lists should now also be clear:

For $(e_1,e_2,\ldots,e_n)$, $n \geq 2$, if t_i is the type of e_i, then the type of the tuple is $(t_1,t_2,\ldots,t_n)$.

For $[e_1,e_2,\ldots,e_n]$, $n \geq 0$, each e_i must have the same type t, and the type of the list is $[t]$.

2.4.1 List Comprehensions and Arithmetic Sequences

As with Lisp dialects, lists are pervasive in Haskell, and as with other functional languages, there is yet more syntactic sugar to aid in their creation. Aside from the constructors for lists just discussed, Haskell provides an expression known as a *list comprehension* that is best explained by example:

```
[ f x | x <- xs ]
```

This expression can intuitively be read as "the list of all `f x` such that `x` is drawn from `xs`." The similarity to set notation is not a coincidence. The phrase `x <- xs` is called a *generator*, of which more than one is allowed, as in:

```
[ (x,y) | x <- xs, y <- ys ]
```

This list comprehension forms the cartesian product of the two lists `xs` and `ys`. The elements are selected as if the generators were "nested" from left to right (with the rightmost generator varying fastest); thus, if `xs` is `[1,2]` and `ys` is `[3,4]`, the result is `[(1,3),(1,4),(2,3),(2,4)]`.

Besides generators, boolean expressions called *guards* are permitted. Guards place constraints on the elements generated. For example, here is a concise definition of everybody's favorite sorting algorithm:

```
quicksort  []           =  []
quicksort (x:xs)        =  quicksort [y | y <- xs, y<x ]
                        ++ [x]
                        ++ quicksort [y | y <- xs, y>=x]
```

To further support the use of lists, Haskell has special syntax for *arithmetic sequences*, which are best explained by a series of examples:

```
[1..10]      ⇒    [1,2,3,4,5,6,7,8,9,10]
[1,3..10]    ⇒    [1,3,5,7,9]
[1,3..]      ⇒    [1,3,5,7,9, ...    (infinite sequence)
```

More will be said about arithmetic sequences in Section 8.2, and "infinite lists" in Section 3.4.

2.4.2 Strings

As another example of syntactic sugar for built-in types, we note that the literal string `"hello"` is actually shorthand for the list of characters `['h','e','l','l','o']`. Indeed, the type of `"hello"` is `String`, where `String` is a predefined type synonym (that we gave as an earlier example):

```
type String             = [Char]
```

This means we can use predefined polymorphic list functions to operate on strings. For example:

```
"hello" ++ " world"   ⇒   "hello world"
```

3 Functions

Since Haskell is a functional language, one would expect functions to play a major role, and indeed they do. In this section, we look at several aspects of functions in Haskell.

First, consider this definition of a function which adds its two arguments:

```
add                     :: Integer -> Integer -> Integer
add x y                 =  x + y
```

This is an example of a *curried* function.[6] An application of `add` has the form `add` e_1 e_2, and is equivalent to `(add` e_1`)` e_2, since function application associates to the *left*. In other words, applying `add` to one argument yields a new function which is then applied to the second argument. This is consistent with the type of `add`, `Integer->Integer->Integer`, which is equivalent to `Integer->(Integer->Integer)`; i.e. `->` associates to the *right*. Indeed, using `add`, we can define `inc` in a different way from earlier:

```
inc                     = add 1
```

This is an example of the *partial application* of a curried function, and is one way that a function can be returned as a value. Let's consider a case in which it's useful to pass a function as an argument. The well-known `map` function is a perfect example:

```
map                     :: (a->b) -> [a] -> [b]
map f  []               =  []
map f (x:xs)            =  f x : map f xs
```

[Function application has higher precedence than any infix operator, and thus the right-hand side of the second equation parses as `(f x) : (map f xs)`.] The `map` function is polymorphic and its type indicates clearly that its first argument is a function; note also that the two `a`'s must be instantiated with the same type (likewise for the `b`'s). As an example of the use of `map`, we can increment the elements in a list:

```
map (add 1) [1,2,3]   ⇒   [2,3,4]
```

[6] The name *curry* derives from the person who popularized the idea: Haskell Curry. To get the effect of an *uncurried* function, we could use a *tuple*, as in:

```
add (x,y)               = x + y
```

But then we see that this version of `add` is really just a function of one argument!

These examples demonstrate the first-class nature of functions, which when used in this way are usually called *higher-order* functions.

3.1 Lambda Abstractions

Instead of using equations to define functions, we can also define them "anonymously" via a *lambda abstraction*. For example, a function equivalent to `inc` could be written as `\x -> x+1`. Similarly, the function `add` is equivalent to `\x -> \y -> x+y`. Nested lambda abstractions such as this may be written using the equivalent shorthand notation `\x y -> x+y`. In fact, the equations:

```
inc x                   = x+1
add x y                 = x+y
```

are really shorthand for:

```
inc                     = \x   -> x+1
add                     = \x y -> x+y
```

We will have more to say about such equivalences later.

In general, given that `x` has type t_1 and `exp` has type t_2, then `\x->exp` has type t_1`->`t_2.

3.2 Infix Operators

Infix operators are really just functions, and can also be defined using equations. For example, here is a definition of a list concatenation operator:

```
(++)                    :: [a] -> [a] -> [a]
[]      ++ ys           =  ys
(x:xs) ++ ys            =  x : (xs++ys)
```

[Lexically, infix operators consist entirely of "symbols," as opposed to normal identifiers which are alphanumeric (§2.4). Haskell has no prefix operators, with the exception of minus (`-`), which is both infix and prefix.]

As another example, an important infix operator on functions is that for *function composition*:

```
(.)                     :: (b->c) -> (a->b) -> (a->c)
f . g                   = \ x -> f (g x)
```

3.2.1 Sections

Since infix operators are really just functions, it makes sense to be able to partially apply them as well. In Haskell the partial application of an infix operator is called a *section*. For example:

```
(x+)     ≡      \y -> x+y
(+y)     ≡      \x -> x+y
(+)      ≡    \x y -> x+y
```

[The parentheses are mandatory.]

The last form of section given above essentially coerces an infix operator into an equivalent functional value, and is handy when passing an infix operator as an argument to a function, as in `map (+) [1,2,3]` (the reader should verify that this returns a list of functions!). It is also necessary when giving a function type signature, as in the examples of `(++)` and `(.)` given earlier.

We can now see that `add` defined earlier is just `(+)`, and `inc` is just `(+1)`! Indeed, these definitions would do just fine:

```
inc                     = (+ 1)
add                     = (+)
```

We can coerce an infix operator into a functional value, but can we go the other way? Yes—we simply enclose an identifier bound to a functional value in backquotes. For example, `x `add` y` is the same as `add x y`.[7] Some functions read better this way. An example is the predefined list membership predicate `elem`; the expression `x `elem` xs` can be read intuitively as "`x` is an element of `xs`."

[There are some special rules regarding sections involving the prefix/infix operator `-`; see (§3.5,§3.4).]

At this point, the reader may be confused at having so many ways to define a function! The decision to provide these mechanisms partly reflects historical conventions, and partly reflects the desire for consistency (for example, in the treatment of infix vs. regular functions).

3.2.2 Fixity Declarations

A *fixity declaration* can be given for any infix operator or constructor (including those made from ordinary identifiers, such as `` `elem` ``). This declaration specifies a precedence level from 0 to 9 (with 9 being the strongest; normal application is assumed to have a precedence level of 10), and left-, right-, or non-associativity. For example, the fixity declarations for `++` and `.` are:

```
infixr 5 ++
infixr 9 .
```

Both of these specify right-associativity, the first with a precedence level of 5, the other 9. Left associativity is specified via `infixl`, and non-associativity by `infix`. Also, the fixity of more than one operator may be specified with the same fixity declaration. If no fixity declaration is given for a particular operator, it defaults to `infixl 9`. (See §5.9 for a detailed definition of the associativity rules.)

3.3 Functions are Non-strict

Suppose `bot` is defined by:

[7] Note carefully that `add` is enclosed in *backquotes*, not *apostrophes* as used in the syntax of characters; i.e. `'f'` is a character, whereas `` `f` `` is an infix operator. Fortunately, most ASCII terminals distinguish these much better than the font used in this manuscript.

```
bot                    = bot
```

In other words, `bot` is a non-terminating expression. Abstractly, we denote the *value* of a non-terminating expression as $\perp$ (read "bottom"). Expressions that result in some kind of a run-time error, such as `1/0`, also have this value. Such an error is not recoverable: programs will not continue past these errors. Errors encountered by the I/O system, such as an end-of-file error, are recoverable and are handled in a different manner. (Such an I/O error is really not an error at all but rather an exception. Much more will be said about exceptions in Section 7.)

A function `f` is said to be *strict* if, when applied to a nonterminating expression, it also fails to terminate. In other words, `f` is strict iff the value of `f bot` is $\perp$. For most programming languages, *all* functions are strict. But this is not so in Haskell. As a simple example, consider `const1`, the constant 1 function, defined by:

```
const1 x               = 1
```

The value of `const1 bot` in Haskell is `1`. Operationally speaking, since `const1` does not "need" the value of its argument, it never attempts to evaluate it, and thus never gets caught in a nonterminating computation. For this reason, non-strict functions are also called "lazy functions", and are said to evaluate their arguments "lazily", or "by need".

Since error and nonterminating values are semantically the same in Haskell, the above argument also holds for errors. For example, `const1 (1/0)` also evaluates properly to `1`.

Non-strict functions are extremely useful in a variety of contexts. The main advantage is that they free the programmer from many concerns about evaluation order. Computationally expensive values may be passed as arguments to functions without fear of them being computed if they are not needed. An important example of this is a possibly *infinite* data structure.

Another way of explaining non-strict functions is that Haskell computes using *definitions* rather than the *assignments* found in traditional languages. Read a declaration such as

```
v                      = 1/0
```

as 'define `v` as `1/0`' instead of 'compute `1/0` and store the result in `v`'. Only if the value (definition) of `v` is needed will the division by zero error occur. By itself, this declaration does not imply any computation. Programming using assignments requires careful attention to the ordering of the assignments: the meaning of the program depends on the order in which the assignments are executed. Definitions, in contrast, are much simpler: they can be presented in any order without affecting the meaning of the program.

3.4 "Infinite" Data Structures

One advantage of the non-strict nature of Haskell is that data constructors are non-strict, too. This should not be surprising, since constructors are really just a special kind of function (the distinguishing feature being that they can be used in pattern matching). For example, the constructor for lists, `(:)`, is non-strict.

Non-strict constructors permit the definition of (conceptually) *infinite* data structures. Here is an infinite list of ones:

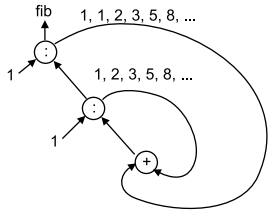


Figure 1: Circular Fibonacci Sequence

```
ones                    = 1 : ones
```

Perhaps more interesting is the function `numsFrom`:

```
numsFrom n              = n : numsFrom (n+1)
```

Thus `numsFrom n` is the infinite list of successive integers beginning with `n`. From it we can construct an infinite list of squares:

```
squares                 = map (^2) (numsfrom 0)
```

(Note the use of a section; `^` is the infix exponentiation operator.)

Of course, eventually we expect to extract some finite portion of the list for actual computation, and there are lots of predefined functions in Haskell that do this sort of thing: `take`, `takeWhile`, `filter`, and others. The definition of Haskell includes a large set of built-in functions and types—this is called the "Standard Prelude". The complete Standard Prelude is included in Appendix A of the Haskell report; see the portion named `PreludeList` for many useful functions involving lists. For example, `take` removes the first `n` elements from a list:

```
take 5 squares    ⇒    [0,1,4,9,16]
```

The definition of `ones` above is an example of a *circular list*. In most circumstances laziness has an important impact on efficiency, since an implementation can be expected to implement the list as a true circular structure, thus saving space.

For another example of the use of circularity, the Fibonacci sequence can be computed efficiently as the following infinite sequence:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

where `zip` is a Standard Prelude function that returns the pairwise interleaving of its two list arguments:

```
zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip  xs     ys          = []
```

Note how `fib`, an infinite list, is defined in terms of itself, as if it were "chasing its tail." Indeed, we can draw a picture of this computation as shown in Figure 1.

For another application of infinite lists, see Section 4.4.

3.5 The Error Function

Haskell has a built-in function called `error` whose type is `String->a`. This is a somewhat odd function: From its type it looks as if it is returning a value of a polymorphic type about which it knows nothing, since it never receives a value of that type as an argument!

In fact, there *is* one value "shared" by all types: $\perp$. Indeed, semantically that is exactly what value is always returned by `error` (recall that all errors have value $\perp$). However, we can expect that a reasonable implementation will print the string argument to `error` for diagnostic purposes. Thus this function is useful when we wish to terminate a program when something has "gone wrong." For example, the actual definition of `head` taken from the Standard Prelude is:

```
head (x:xs)             =  x
head  []                =  error "head{PreludeList}: head []"
```

4 Case Expressions and Pattern Matching

Earlier we gave several examples of pattern matching in defining functions—for example `length` and `fringe`. In this section we will look at the pattern-matching process in greater detail (§3.17).[8]

Patterns are not "first-class;" there is only a fixed set of different kinds of patterns. We have already seen several examples of *data constructor* patterns; both `length` and `fringe` defined earlier use such patterns, the former on the constructors of a "built-in" type (lists), the latter on a user-defined type (`Tree`). Indeed, matching is permitted using the constructors of any type, user-defined or not. This includes tuples, strings, numbers, characters, etc. For example, here's a contrived function that matches against a tuple of "constants:"

```
contrived :: ([a], Char, (Int, Float), String, Bool) -> Bool
contrived    ([],  'b',  (1,   2.0),   "hi",   True) = False
```

This example also demonstrates that *nesting* of patterns is permitted (to arbitrary depth).

Technically speaking, *formal parameters*[9] are also patterns—it's just that they *never fail to match a value*. As a "side effect" of the successful match, the formal parameter is bound to the value it is being matched against. For this reason patterns in any one equation are not allowed to have more than one occurrence of the same formal parameter (a property called *linearity* §3.17, §3.3, §4.4.2).

Patterns such as formal parameters that never fail to match are said to be *irrefutable*, in contrast to *refutable* patterns which may fail to match. The pattern used in the `contrived` example above is refutable. There are three other kinds of irrefutable patterns, two of which we will introduce now (the other we will delay until Section 4.4).

[8] Pattern matching in Haskell is different from that found in logic programming languages such as Prolog; in particular, it can be viewed as "one-way" matching, whereas Prolog allows "two-way" matching (via unification), along with implicit backtracking in its evaluation mechanism.

[9] The Report calls these *variables*.

As-patterns. Sometimes it is convenient to name a pattern for use on the right-hand side of an equation. For example, a function that duplicates the first element in a list might be written as:

```
f (x:xs)                = x:x:xs
```

(Recall that ":" associates to the right.) Note that `x:xs` appears both as a pattern on the left-hand side, and an expression on the right-hand side. To improve readability, we might prefer to write `x:xs` just once, which we can achieve using an *as-pattern* as follows:[10]

```
f s@(x:xs)              = x:s
```

Technically speaking, as-patterns always result in a successful match, although the sub-pattern (in this case `x:xs`) could, of course, fail.

Wild-cards. Another common situation is matching against a value we really care nothing about. For example, the functions `head` and `tail` defined in Section 2.1 can be rewritten as:

```
head (x:_)              = x
tail (_:xs)             = xs
```

in which we have "advertised" the fact that we don't care what a certain part of the input is. Each wild-card independently matches anything, but in contrast to a formal parameter, each binds nothing; for this reason more than one is allowed in an equation.

4.1 Pattern-Matching Semantics

So far we have discussed how individual patterns are matched, how some are refutable, some are irrefutable, etc. But what drives the overall process? In what order are the matches attempted? What if none succeeds? This section addresses these questions.

Pattern matching can either *fail*, *succeed* or *diverge*. A successful match binds the formal parameters in the pattern. Divergence occurs when a value needed by the pattern contains an error ($\perp$). The matching process itself occurs "top-down, left-to-right." Failure of a pattern anywhere in one equation results in failure of the whole equation, and the next equation is then tried. If all equations fail, the value of the function application is $\perp$, and results in a run-time error.

For example, if `[1,2]` is matched against `[0,bot]`, then `1` fails to match `0`, so the result is a failed match. (Recall that `bot`, defined earlier, is a variable bound to $\perp$.) But if `[1,2]` is matched against `[bot,0]`, then matching `1` against `bot` causes divergence (i.e. $\perp$).

The other twist to this set of rules is that top-level patterns may also have a boolean *guard*, as in this definition of a function that forms an abstract version of a number's sign:

```
sign x |  x >  0        =   1
       |  x == 0        =   0
       |  x <  0        =  -1
```

Note that a sequence of guards may be provided for the same pattern; as with patterns, they are evaluated top-down, and the first that evaluates to `True` results in a successful match.

[10] Another advantage to doing this is that a naive implementation might completely reconstruct `x:xs` rather than re-use the value being matched against.

4.2 An Example

The pattern-matching rules can have subtle effects on the meaning of functions. For example, consider this definition of `take`:

```
take  0     _          =  []
take  _     []         =  []
take  n     (x:xs)     =  x : take (n-1) xs
```

and this slightly different version (the first 2 equations have been reversed):

```
take1  _     []        =  []
take1  0     _         =  []
take1  n     (x:xs)    =  x : take1 (n-1) xs
```

Now note the following:

```
take  0 bot     ⇒    []
take1 0 bot     ⇒    ⊥

take  bot []    ⇒    ⊥
take1 bot []    ⇒    []
```

We see that `take` is "more defined" with respect to its second argument, whereas `take1` is more defined with respect to its first. It is difficult to say in this case which definition is better. Just remember that in certain applications, it may make a difference. (The Standard Prelude includes a definition corresponding to `take`.)

4.3 Case Expressions

Pattern matching provides a way to "dispatch control" based on structural properties of a value. In many circumstances we don't wish to define a *function* every time we need to do this, but so far we have only shown how to do pattern matching in function definitions. Haskell's *case expression* provides a way to solve this problem. Indeed, the meaning of pattern matching in function definitions is specified in the Report in terms of case expressions, which are considered more primitive. In particular, a function definition of the form:

$$\begin{array}{l}\texttt{f}\ p_{11}\ \ldots\ p_{1k}\ \texttt{=}\ e_1\\ \ldots\\ \texttt{f}\ p_{n1}\ \ldots\ p_{nk}\ \texttt{=}\ e_n\end{array}$$

where each p_{ij} is a pattern, is semantically equivalent to:

$$\begin{array}{l}\texttt{f x1 x2 ... xk = case (x1, ..., xk) of}\ (p_{11}, \ldots, p_{1k})\ \texttt{->}\ e_1\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_{n1}, \ldots, p_{nk})\ \texttt{->}\ e_n\end{array}$$

where the `xi` are new identifiers. (For a more general translation that includes guards, see §4.4.2.) For example, the definition of `take` given earlier is equivalent to:

```
take m ys               = case (m,ys) of
                            (0,_)       ->  []
                            (_,[])      ->  []
                            (n,x:xs)    ->  x : take (n-1) xs
```

A point not made earlier is that, for type correctness, the types of the right-hand sides of a case expression or set of equations comprising a function definition must all be the same; more precisely, they must all share a common principal type.

The pattern-matching rules for case expressions are the same as we have given for function definitions, so there is really nothing new to learn here, other than to note the convenience that case expressions offer. Indeed, there's one use of a case expression that is so common that it has special syntax: the *conditional expression*. In Haskell, conditional expressions have the familiar form:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

which is really short-hand for:

$$\begin{array}{l}\texttt{case } e_1 \texttt{ of } \quad \texttt{True} \ \texttt{->} \ e_2 \\ \qquad\qquad\quad\ \ \texttt{False} \ \texttt{->} \ e_3\end{array}$$

From this expansion it should be clear that e_1 must have type `Bool`, and e_2 and e_3 must have the same (but otherwise arbitrary) type. In other words, `if-then-else` when viewed as a function has type `Bool->a->a->a`.

4.4 Lazy Patterns

There is one other kind of pattern allowed in Haskell. It is called a *lazy pattern*, and has the form *~pat*. Lazy patterns are *irrefutable*: matching a value v against *~pat* always succeeds, regardless of *pat*. Operationally speaking, if an identifier in *pat* is later "used" on the right-hand-side, it will be bound to that portion of the value that would result if v were to successfully match *pat*, and $\perp$ otherwise.

Lazy patterns are useful in contexts where infinite data structures are being defined recursively. For example, infinite lists are an excellent vehicle for writing *simulation* programs, and in this context the infinite lists are often called *streams*. Consider the simple case of simulating the interactions between a server process `server` and a client process `client`, where `client` sends a sequence of *requests* to `server`, and `server` replies to each request with some kind of *response*. This situation is shown pictorially in Figure 2. (Note that `client` also takes an initial message as argument.) Using streams to simulate the message sequences, the Haskell code corresponding to this diagram is:

```
reqs                    = client init resps
resps                   = server reqs
```

These recursive equations are a direct lexical transliteration of the diagram.

Let us further assume that the structure of the server and client look something like this:

```
client init (resp:resps) = init : client (next resp) resps
server      (req:reqs)   = process req : server reqs
```

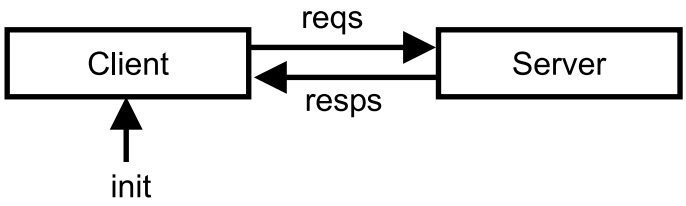


Figure 2: Client-Server Simulation

where we assume that `next` is a function that, given a response from the server, determines the next request, and `process` is a function that processes a request from the client, returning an appropriate response.

Unfortunately, this program has a serious problem: it will not produce any output! The problem is that `client`, as used in the recursive setting of `reqs` and `resps`, attempts a match on the response list before it has submitted its first request! In other words, the pattern matching is being done "too early." One way to fix this is to redefine `client` as follows:

```
client init resps          = init : client (next (head resps)) (tail resps)
```

Although workable, this solution does not read as well as that given earlier. A better solution is to use a lazy pattern:

```
client init ~(resp:resps) = init : client (next resp) resps
```

Because lazy patterns are irrefutable, the match will immediately succeed, allowing the initial request to be "submitted", in turn allowing the first response to be generated; the engine is now "primed", and the recursion takes care of the rest.

As an example of this program in action, if we define:

```
init                    = 0
next resp               = resp
process req             = req+1
```

then we see that:

```
take 10 reqs    ⇒    [0,1,2,3,4,5,6,7,8,9]
```

As another example of the use of lazy patterns, consider the definition of Fibonacci given earlier:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

We might try rewriting this using an as-pattern:

```
fib@(1:tfib)    = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

This version of `fib` has the (small) advantage of not using `tail` on the right-hand side, since it is available in "destructured" form on the left-hand side as `tfib`.

[This kind of equation is called a *pattern binding* because it is a top-level equation in which the entire left-hand side is a pattern; i.e. both `fib` and `tfib` become bound within the scope of the declaration.]

Now, using the same reasoning as earlier, we should be led to believe that this program will not generate any output. Curiously, however, it *does*, and the reason is simple: in Haskell, pattern bindings are assumed to have an implicit ~ in front of them, reflecting the most common behavior expected of pattern bindings, and avoiding some anomalous situations which are beyond the scope of this tutorial. Thus we see that lazy patterns play an important role in Haskell, if only implicitly.

4.5 Lexical Scoping and Nested Forms

It is often desirable to create a nested scope within an expression, for the purpose of creating local bindings not seen elsewhere—i.e. some kind of "block-structuring" form. In Haskell there are two ways to achieve this:

Let Expressions. Haskell's *let expressions* are useful whenever a nested set of bindings is required. As a simple example, consider:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

The set of bindings created by a `let` expression is *mutually recursive*, and pattern bindings are treated as lazy patterns (i.e. they carry an implicit ~). The only kind of declarations permitted are *type signatures*, *function bindings*, and *pattern bindings*.

Where Clauses. Sometimes it is convenient to scope bindings over several guarded equations, which requires a *where clause*:

```
f x y  |  y>z            =  ...
       |  y==z           =  ...
       |  y<z            =  ...
     where z = x*x
```

Note that this cannot be done with a `let` expression, which only scopes over the expression which it encloses. A `where` clause is only allowed at the top level of a set of equations or case expression. The same properties and constraints on bindings in `let` expressions apply to those in `where` clauses.

These two forms of nested scope seem very similar, but remember that a `let` expression is an *expression*, whereas a `where` clause is not—it is part of the syntax of function declarations and case expressions.

4.6 Layout

The reader may have been wondering how it is that Haskell programs avoid the use of semicolons, or some other kind of terminator, to mark the end of equations, declarations, etc. For example, consider this `let` expression from the last section:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

How does the parser know not to parse this as:

```
let y   = a*b f
    x   = (x+y)/y
in f c + f d
```

?

The answer is that Haskell uses a two-dimensional syntax called *layout* that essentially relies on declarations being "lined up in columns." In the above example, note that `y` and `f` begin in the same column. The rules for layout are spelled out in detail in the Report (§2.7, §B.3), but in practice, use of layout is rather intuitive. Just remember two things:

First, the next character following any of the keywords `where`, `let`, or `of` is what determines the starting column for the declarations in the where, let, or case expression being written (the rule also applies to `where` used in the class and instance declarations to be introduced in Section 5). Thus we can begin the declarations on the same line as the keyword, the next line, etc. (The `do` keyword, to be discussed later, also uses layout).

Second, just be sure that the starting column is further to the right than the starting column associated with the immediately surrounding clause (otherwise it would be ambiguous). The "termination" of a declaration happens when something appears at or to the left of the starting column associated with that binding form.[11]

Layout is actually shorthand for an *explicit* grouping mechanism, which deserves mention because it can be useful under certain circumstances. The `let` example above is equivalent to:

```
let { y   = a*b
    ; f x = (x+y)/y
    }
in f c + f d
```

Note the explicit curly braces and semicolons. One way in which this explicit notation is useful is when more than one declaration is desired on a line; for example, this is a valid expression:

```
let y   = a*b;  z = a/b
    f x = (x+y)/z
in f c + f d
```

For another example of the expansion of layout into explicit delimiters, see §2.7.

The use of layout greatly reduces the syntactic clutter associated with declaration lists, thus enhancing readability. It is easy to learn, and its use is encouraged.

5 Type Classes and Overloading

There is one final feature of Haskell's type system that sets it apart from other programming languages. The kind of polymorphism that we have talked about so far is commonly called *parametric* polymorphism. There is another kind called *ad hoc* polymorphism, better known as *overloading*. Here are some examples of *ad hoc* polymorphism:

[11] Haskell observes the convention that tabs count as 8 blanks; thus care must be taken when using an editor which may observe some other convention.

- The literals `1`, `2`, etc. are often used to represent both fixed and arbitrary precision integers.
- Numeric operators such as `+` are often defined to work on many different kinds of numbers.
- The equality operator (`==` in Haskell) usually works on numbers and many other (but not all) types.

Note that these overloaded behaviors are different for each type (in fact the behavior is sometimes undefined, or error), whereas in parametric polymorphism the type truly does not matter (`fringe`, for example, really doesn't care what kind of elements are found in the leaves of a tree). In Haskell, *type classes* provide a structured way to control *ad hoc* polymorphism, or overloading.

Let's start with a simple, but important, example: equality. There are many types for which we would like equality defined, but some for which we would not. For example, comparing the equality of functions is generally considered computationally intractable, whereas we often want to compare two lists for equality.[12] To highlight the issue, consider this definition of the function `elem` which tests for membership in a list:

```
x `elem`  []            = False
x `elem` (y:ys)         = x==y || (x `elem` ys)
```

[For the stylistic reason we discussed in Section 3.1, we have chosen to define `elem` in infix form. `==` and `||` are the infix operators for equality and logical or, respectively.]

Intuitively speaking, the type of `elem` "ought" to be: `a->[a]->Bool`. But this would imply that `==` has type `a->a->Bool`, even though we just said that we don't expect `==` to be defined for all types.

Furthermore, as we have noted earlier, even if `==` were defined on all types, comparing two lists for equality is very different from comparing two integers. In this sense, we expect `==` to be *overloaded* to carry on these various tasks.

Type classes conveniently solve both of these problems. They allow us to declare which types are *instances* of which class, and to provide definitions of the overloaded *operations* associated with a class. For example, let's define a type class containing an equality operator:

```
class Eq a where
  (==)                  :: a -> a -> Bool
```

Here `Eq` is the name of the class being defined, and `==` is the single operation in the class. This declaration may be read "a type `a` is an instance of the class `Eq` if there is an (overloaded) operation `==`, of the appropriate type, defined on it." (Note that `==` is only defined on pairs of objects of the same type.)

The constraint that a type `a` must be an instance of the class `Eq` is written `Eq a`. Thus `Eq a` is not a type expression, but rather it expresses a constraint on a type, and is called a *context*. Contexts are placed at the front of type expressions. For example, the effect of the above class declaration is to assign the following type to `==`:

```
(==)                    :: (Eq a) => a -> a -> Bool
```

[12] The kind of equality we are referring to here is "value equality," and opposed to the "pointer equality" found, for example, with Java's `==`. Pointer equality is not referentially transparent, and thus does not sit well in a purely functional language.

This should be read, "For every type `a` that is an instance of the class `Eq`, `==` has type `a->a->Bool`". This is the type that would be used for `==` in the `elem` example, and indeed the constraint imposed by the context propagates to the principal type for `elem`:

```
elem                    :: (Eq a) => a -> [a] -> Bool
```

This is read, "For every type `a` that is an instance of the class `Eq`, `elem` has type `a->[a]->Bool`". This is just what we want—it expresses the fact that `elem` is not defined on all types, just those for which we know how to compare elements for equality.

So far so good. But how do we specify which types are instances of the class `Eq`, and the actual behavior of `==` on each of those types? This is done with an *instance declaration*. For example:

```
instance Eq Integer where
  x == y                =  x `integerEq` y
```

The definition of `==` is called a *method*. The function `integerEq` happens to be the primitive function that compares integers for equality, but in general any valid expression is allowed on the right-hand side, just as for any other function definition. The overall declaration is essentially saying: "The type `Integer` is an instance of the class `Eq`, and here is the definition of the method corresponding to the operation `==`." Given this declaration, we can now compare fixed precision integers for equality using `==`. Similarly:

```
instance Eq Float where
  x == y                =  x `floatEq` y
```

allows us to compare floating point numbers using `==`.

Recursive types such as `Tree` defined earlier can also be handled:

```
instance (Eq a) => Eq (Tree a) where
  Leaf a         == Leaf b          =  a == b
  (Branch l1 r1) == (Branch l2 r2)  =  (l1==l2) && (r1==r2)
  _              == _               =  False
```

Note the context `Eq a` in the first line—this is necessary because the elements in the leaves (of type `a`) are compared for equality in the second line. The additional constraint is essentially saying that we can compare trees of `a`'s for equality as long as we know how to compare `a`'s for equality. If the context were omitted from the instance declaration, a static type error would result.

The Haskell Report, especially the Prelude, contains a wealth of useful examples of type classes. Indeed, a class `Eq` is defined that is slightly larger than the one defined earlier:

```
class  Eq a  where
  (==), (/=)            :: a -> a -> Bool
  x /= y                =  not (x == y)
```

This is an example of a class with two operations, one for equality, the other for inequality. It also demonstrates the use of a *default method*, in this case for the inequality operation `/=`. If a method for a particular operation is omitted in an instance declaration, then the default one defined in the class declaration, if it exists, is used instead. For example, the three instances of `Eq` defined earlier will work perfectly well with the above class declaration, yielding just the right definition of inequality that we want: the logical negation of equality.

Haskell also supports a notion of *class extension*. For example, we may wish to define a class `Ord` which *inherits* all of the operations in `Eq`, but in addition has a set of comparison operations and minimum and maximum functions:

```
class  (Eq a) => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

Note the context in the `class` declaration. We say that `Eq` is a *superclass* of `Ord` (conversely, `Ord` is a *subclass* of `Eq`), and any type which is an instance of `Ord` must also be an instance of `Eq`. (In the next Section we give a fuller definition of `Ord` taken from the Prelude.)

One benefit of such class inclusions is shorter contexts: a type expression for a function that uses operations from both the `Eq` and `Ord` classes can use the context `(Ord a)`, rather than `(Eq a, Ord a)`, since `Ord` "implies" `Eq`. More importantly, methods for subclass operations can assume the existence of methods for superclass operations. For example, the `Ord` declaration in the Standard Prelude contains this default method for `(<)`:

```
    x < y               =  x <= y && x /= y
```

As an example of the use of `Ord`, the principal typing of `quicksort` defined in Section 2.4.1 is:

```
quicksort               ::  (Ord a) => [a] -> [a]
```

In other words, `quicksort` only operates on lists of values of ordered types. This typing for `quicksort` arises because of the use of the comparison operators `<` and `>=` in its definition.

Haskell also permits *multiple inheritance*, since classes may have more than one superclass. For example, the declaration

```
class (Eq a, Show a) => C a where ...
```

creates a class `C` which inherits operations from both `Eq` and `Show`.

Class methods are treated as top level declarations in Haskell. They share the same namespace as ordinary variables; a name cannot be used to denote both a class method and a variable or methods in different classes.

Contexts are also allowed in `data` declarations; see §4.2.1.

Class methods may have additional class constraints on any type variable except the one defining the current class. For example, in this class:

```
class C a where
  m                     :: Show b => a -> b
```

the method `m` requires that type `b` is in class `Show`. However, the method `m` could not place any additional class constraints on type `a`. These would instead have to be part of the context in the class declaration.

So far, we have been using "first-order" types. For example, the type constructor `Tree` has so far always been paired with an argument, as in `Tree Integer` (a tree containing `Integer` values) or `Tree a` (representing the family of trees containing `a` values). But `Tree` by itself is a type constructor, and as such takes a type as an argument and returns a type as a result. There are

no values in Haskell that have this type, but such "higher-order" types can be used in `class` declarations.

To begin, consider the following `Functor` class (taken from the Prelude):

```
class Functor f where
  fmap                  :: (a -> b) -> f a -> f b
```

The `fmap` function generalizes the `map` function used previously. Note that the type variable `f` is applied to other types in `f a` and `f b`. Thus we would expect it to be bound to a type such as `Tree` which can be applied to an argument. An instance of `Functor` for type `Tree` would be:

```
instance Functor Tree where
  fmap f (Leaf x)       = Leaf   (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

This instance declaration declares that `Tree`, rather than `Tree a`, is an instance of `Functor`. This capability is quite useful, and here demonstrates the ability to describe generic "container" types, allowing functions such as `fmap` to work uniformly over arbitrary trees, lists, and other data types.

[Type applications are written in the same manner as function applications. The type `T a b` is parsed as `(T a) b`. Types such as tuples which use special syntax can be written in an alternative style which allows currying. For functions, `(->)` is a type constructor; the types `f -> g` and `(->) f g` are the same. Similarly, the types `[a]` and `[] a` are the same. For tuples, the type constructors (as well as the data constructors) are `(,)`, `(,,)`, and so on.]

As we know, the type system detects typing errors in expressions. But what about errors due to malformed type expressions? The expression `(+) 1 2 3` results in a type error since `(+)` takes only two arguments. Similarly, the type `Tree Int Int` should produce some sort of an error since the `Tree` type takes only a single argument. So, how does Haskell detect malformed type expressions? The answer is a second type system which ensures the correctness of types! Each type has an associated *kind* which ensures that the type is used correctly.

Type expressions are classified into different *kinds* which take one of two possible forms:

- The symbol $*$ represents the kind of type associated with concrete data objects. That is, if the value v has type t, the kind of v must be $*$.
- If κ_1 and κ_2 are kinds, then $\kappa_1 \to \kappa_2$ is the kind of types that take a type of kind κ_1 and return a type of kind κ_2.

The type constructor `Tree` has the kind $* \to *$; the type `Tree Int` has the kind $*$. Members of the `Functor` class must all have the kind $* \to *$; a kinding error would result from an declaration such as

```
instance Functor Integer where ...
```

since `Integer` has the kind $*$.

Kinds do not appear directly in Haskell programs. The compiler infers kinds before doing type checking without any need for 'kind declarations'. Kinds stay in the background of a Haskell program except when an erroneous type signature leads to a kind error. Kinds are simple enough that compilers should be able to provide descriptive error messages when kind conflicts occur. See §4.1.1 and §4.6 for more information about kinds.

A Different Perspective. Before going on to further examples of the use of type classes, it is worth pointing out two other views of Haskell's type classes. The first is by analogy with object-oriented programming (OOP). In the following general statement about OOP, simply substituting *type class* for *class*, and *type* for *object*, yields a valid summary of Haskell's type class mechanism:

"*Classes* capture common sets of operations. A particular *object* may be an instance of a *class*, and will have a method corresponding to each operation. *Classes* may be arranged hierarchically, forming notions of super*classes* and sub*classes*, and permitting inheritance of operations/methods. A default method may also be associated with an operation."

In contrast to OOP, it should be clear that types are not objects, and in particular there is no notion of an object's or type's internal mutable state. An advantage over some OOP languages is that methods in Haskell are completely type-safe: any attempt to apply a method to a value whose type is not in the required class will be detected at compile time instead of at runtime. In other words, methods are not "looked up" at runtime but are simply passed as higher-order functions.

A different perspective can be gotten by considering the relationship between parametric and *ad hoc* polymorphism. We have shown how parametric polymorphism is useful in defining families of types by universally quantifying over all types. Sometimes, however, that universal quantification is too broad—we wish to quantify over some smaller set of types, such as those types whose elements can be compared for equality. Type classes can be seen as providing a structured way to do just this. Indeed, we can think of parametric polymorphism as a kind of overloading too! It's just that the overloading occurs implicitly over all types instead of a constrained set of types (i.e. a type class).

Comparison to Other Languages. The classes used by Haskell are similar to those used in other object-oriented languages such as C++ and Java. However, there are some significant differences:

- Haskell separates the definition of a type from the definition of the methods associated with that type. A class in C++ or Java usually defines both a data structure (the member variables) and the functions associated with the structure (the methods). In Haskell, these definitions are separated.
- The class methods defined by a Haskell class correspond to virtual functions in a C++ class. Each instance of a class provides its own definition for each method; class defaults correspond to default definitions for a virtual function in the base class.
- Haskell classes are roughly similar to a Java interface. Like an interface declaration, a Haskell class declaration defines a protocol for using an object rather than defining an object itself.
- Haskell does not support the C++ overloading style in which functions with different types share a common name.
- The type of a Haskell object cannot be implicitly coerced; there is no universal base class such as `Object` which values can be projected into or out of.

- C++ and Java attach identifying information (such as a VTable) to the runtime representation of an object. In Haskell, such information is attached logically instead of physically to values, through the type system.
- There is no access control (such as public or private class constituents) built into the Haskell class system. Instead, the module system must be used to hide or reveal components of a class.

6 Types, Again

Here we examine some of the more advanced aspects of type declarations.

6.1 The Newtype Declaration

A common programming practice is to define a type whose representation is identical to an existing one but which has a separate identity in the type system. In Haskell, the `newtype` declaration creates a new type from an existing one. For example, natural numbers can be represented by the type `Integer` using the following declaration:

```
newtype Natural = MakeNatural Integer
```

This creates an entirely new type, `Natural`, whose only constructor contains a single `Integer`. The constructor `MakeNatural` converts between an `Natural` and an `Integer`:

```
toNatural               :: Integer -> Natural
toNatural x | x < 0     = error "Can't create negative naturals!"
            | otherwise = MakeNatural x

fromNatural             :: Natural -> Integer
fromNatural (MakeNatural i) = i
```

The following instance declaration admits `Natural` to the `Num` class:

```
instance Num Natural where
    fromInteger         = toNatural
    x + y               = toNatural (fromNatural x + fromNatural y)
    x - y               = let r = fromNatural x - fromNatural y in
                              if r < 0 then error "Unnatural subtraction"
                                       else toNatural r
    x * y               = toNatural (fromNatural x * fromNatural y)
```

Without this declaration, `Natural` would not be in `Num`. Instances declared for the old type do not carry over to the new one. Indeed, the whole purpose of this type is to introduce a different `Num` instance. This would not be possible if `Natural` were defined as a type synonym of `Integer`.

All of this works using a `data` declaration instead of a `newtype` declaration. However, the `data` declaration incurs extra overhead in the representation of `Natural` values. The use of `newtype` avoids the extra level of indirection (caused by laziness) that the `data` declaration would introduce.

See section 4.2.3 of the report for a more discussion of the relation between `newtype`, `data`, and `type` declarations.

[Except for the keyword, the `newtype` declaration uses the same syntax as a `data` declaration with a single constructor containing a single field. This is appropriate since types defined using `newtype` are nearly identical to those created by an ordinary `data` declaration.]

6.2 Field Labels

The fields within a Haskell data type can be accessed either positionally or by name using *field labels*. Consider a data type for a two-dimensional point:

```
data Point = Pt Float Float
```

The two components of a `Point` are the first and second arguments to the constructor `Pt`. A function such as

```
pointx                  :: Point -> Float
pointx (Pt x _)         =  x
```

may be used to refer to the first component of a point in a more descriptive way, but, for large structures, it becomes tedious to create such functions by hand.

Constructors in a `data` declaration may be declared with associated *field names*, enclosed in braces. These field names identify the components of constructor by name rather than by position. This is an alternative way to define `Point`:

```
data Point = Pt {pointx, pointy :: Float}
```

This data type is identical to the earlier definition of `Point`. The constructor `Pt` is the same in both cases. However, this declaration also defines two field names, `pointx` and `pointy`. These field names can be used as *selector functions* to extract a component from a structure. In this example, the selectors are:

```
pointx                  ::   Point -> Float
pointy                  ::   Point -> Float
```

This is a function using these selectors:

```
absPoint                :: Point -> Float
absPoint p              =  sqrt (pointx p * pointx p +
                                 pointy p * pointy p)
```

Field labels can also be used to construct new values. The expression `Pt {pointx=1, pointy=2}` is identical to `Pt 1 2`. The use of field names in the declaration of a data constructor does not preclude the positional style of field access; both `Pt {pointx=1, pointy=2}` and `Pt 1 2` are allowed. When constructing a value using field names, some fields may be omitted; these absent fields are undefined.

Pattern matching using field names uses a similar syntax for the constructor `Pt`:

```
absPoint (Pt {pointx = x, pointy = y}) = sqrt (x*x + y*y)
```

An update function uses field values in an existing structure to fill in components of a new structure. If `p` is a `Point`, then `p {pointx=2}` is a point with the same `pointy` as `p` but with `pointx` replaced by `2`. This is not a destructive update: the update function merely creates a new copy of the object, filling in the specified fields with new values.

[The braces used in conjunction with field labels are somewhat special: Haskell syntax usually allows braces to be omitted using the *layout rule* (described in Section 4.6). However, the braces associated with field names must be explicit.]

Field names are not restricted to types with a single constructor (commonly called 'record' types). In a type with multiple constructors, selection or update operations using field names may fail at runtime. This is similar to the behavior of the `head` function when applied to an empty list.

Field labels share the top level namespace with ordinary variables and class methods. A field name cannot be used in more than one data type in scope. However, within a data type, the same field name can be used in more than one of the constructors so long as it has the same typing in all cases. For example, in this data type

```
data T = C1 {f :: Int, g :: Float}
       | C2 {f :: Int, h :: Bool}
```

the field name `f` applies to both constructors in `T`. Thus if `x` is of type `T`, then `x {f=5}` will work for values created by either of the constructors in `T`.

Field names does not change the basic nature of an algebraic data type; they are simply a convenient syntax for accessing the components of a data structure by name rather than by position. They make constructors with many components more manageable since fields can be added or removed without changing every reference to the constructor. For full details of field labels and their semantics, see Section §4.2.1.

6.3 Strict Data Constructors

Data structures in Haskell are generally *lazy*: the components are not evaluated until needed. This permits structures that contain elements which, if evaluated, would lead to an error or fail to terminate. Lazy data structures enhance the expressiveness of Haskell and are an essential aspect of the Haskell programming style.

Internally, each field of a lazy data object is wrapped up in a structure commonly referred to as a *thunk* that encapsulates the computation defining the field value. This thunk is not entered until the value is needed; thunks which contain errors ($\perp$) do not affect other elements of a data structure. For example, the tuple `('a',`$\perp$`)` is a perfectly legal Haskell value. The `'a'` may be used without disturbing the other component of the tuple. Most programming languages are *strict* instead of lazy: that is, all components of a data structure are reduced to values before being placed in the structure.

There are a number of overheads associated with thunks: they take time to construct and evaluate, they occupy space in the heap, and they cause the garbage collector to retain other structures needed for the evaluation of the thunk. To avoid these overheads, *strictness flags* in

`data` declarations allow specific fields of a constructor to be evaluated immediately, selectively suppressing laziness. A field marked by `!` in a `data` declaration is evaluated when the structure is created instead of delayed in a thunk.

There are a number of situations where it may be appropriate to use strictness flags:

- Structure components that are sure to be evaluated at some point during program execution.
- Structure components that are simple to evaluate and never cause errors.
- Types in which partially undefined values are not meaningful.

For example, the complex number library defines the `Complex` type as:

```
data RealFloat a => Complex a = !a :+ !a
```

[note the infix definition of the constructor `:+`.] This definition marks the two components, the real and imaginary parts, of the complex number as being strict. This is a more compact representation of complex numbers but this comes at the expense of making a complex number with an undefined component, `1 :+` $\perp$ for example, totally undefined ($\perp$). As there is no real need for partially defined complex numbers, it makes sense to use strictness flags to achieve a more efficient representation.

Strictness flags may be used to address memory leaks: structures retained by the garbage collector but no longer necessary for computation.

The strictness flag, `!`, can only appear in `data` declarations. It cannot be used in other type signatures or in any other type definitions. There is no corresponding way to mark function arguments as being strict, although the same effect can be obtained using the `seq` or `!$` functions. See §4.2.1 for further details.

It is difficult to present exact guidelines for the use of strictness flags. They should be used with caution: laziness is one of the fundamental properties of Haskell and adding strictness flags may lead to hard to find infinite loops or have other unexpected consequences.

7 Input/Output

The I/O system in Haskell is purely functional, yet has all of the expressive power found in conventional programming languages. In imperative languages, programs proceed via *actions* which examine and modify the current state of the world. Typical actions include reading and setting global variables, writing files, reading input, and opening windows. Such actions are also a part of Haskell but are cleanly separated from the purely functional core of the language.

Haskell's I/O system is built around a somewhat daunting mathematical foundation: the *monad*. However, understanding of the underlying monad theory is not necessary to program using the I/O system. Rather, monads are a conceptual structure into which I/O happens to fit. It is no more necessary to understand monad theory to perform Haskell I/O than it is to understand group theory to do simple arithmetic. A detailed explanation of monads is found in Section 9.

The monadic operators that the I/O system is built upon are also used for other purposes; we will look more deeply into monads later. For now, we will avoid the term monad and concentrate on the use of the I/O system. It's best to think of the I/O monad as simply an abstract data type.

Actions are defined rather than invoked within the expression language of Haskell. Evaluating the definition of an action doesn't actually cause the action to happen. Rather, the invocation of actions takes place outside of the expression evaluation we have considered up to this point.

Actions are either atomic, as defined in system primitives, or are a sequential composition of other actions. The I/O monad contains primitives which build composite actions, a process similar to joining statements in sequential order using ';' in other languages. Thus the monad serves as the glue which binds together the actions in a program.

7.1 Basic I/O Operations

Every I/O action returns a value. In the type system, the return value is 'tagged' with `IO` type, distinguishing actions from other values. For example, the type of the function `getChar` is:

```
getChar                 ::   IO Char
```

The `IO Char` indicates that `getChar`, when invoked, performs some action which returns a character. Actions which return no interesting values use the unit type, `()`. For example, the `putChar` function:

```
putChar                 ::    Char -> IO ()
```

takes a character as an argument but returns nothing useful. The unit type is similar to `void` in other languages.

Actions are sequenced using an operator that has a rather cryptic name: `>>=` (or 'bind'). Instead of using this operator directly, we choose some syntactic sugar, the `do` notation, to hide these sequencing operators under a syntax resembling more conventional languages. The `do` notation can be trivially expanded to `>>=`, as described in §3.14.

The keyword `do` introduces a sequence of statements which are executed in order. A statement is either an action, a pattern bound to the result of an action using `<-`, or a set of local definitions introduced using `let`. The `do` notation uses layout in the same manner as `let` or `where` so we can omit braces and semicolons with proper indentation. Here is a simple program to read and then print a character:

```
main                    :: IO ()
main                    =  do c <- getChar
                              putChar c
```

The use of the name `main` is important: `main` is defined to be the entry point of a Haskell program (similar to the `main` function in C), and must have an `IO` type, usually `IO ()`. (The name `main` is special only in the module `Main`; we will have more to say about modules later.) This program performs two actions in sequence: first it reads in a character, binding the result to the variable c, and then prints the character. Unlike a `let` expression where variables are scoped over all definitions, the variables defined by `<-` are only in scope in the following statements.

There is still one missing piece. We can invoke actions and examine their results using `do`, but how do we return a value from a sequence of actions? For example, consider the `ready` function that reads a character and returns `True` if the character was a 'y':

```
ready                   :: IO Bool
ready                   =  do c <- getChar
                              c == 'y'  -- Bad!!!
```

This doesn't work because the second statement in the 'do' is just a boolean value, not an action. We need to take this boolean and create an action that does nothing but return the boolean as its result. The `return` function does just that:

```
return                  ::   a -> IO a
```

The `return` function completes the set of sequencing primitives. The last line of `ready` should read `return (c == 'y')`.

We are now ready to look at more complicated I/O functions. First, the function `getLine`:

```
getLine     :: IO String
getLine     =  do c <- getChar
                  if c == '\n'
                       then return ""
                       else do l <- getLine
                               return (c:l)
```

Note the second `do` in the else clause. Each `do` introduces a single chain of statements. Any intervening construct, such as the `if`, must use a new `do` to initiate further sequences of actions.

The `return` function admits an ordinary value such as a boolean to the realm of I/O actions. What about the other direction? Can we invoke some I/O actions within an ordinary expression? For example, how can we say `x + print y` in an expression so that `y` is printed out as the expression evaluates? The answer is that we can't! It is *not* possible to sneak into the imperative universe while in the midst of purely functional code. Any value 'infected' by the imperative world must be tagged as such. A function such as

```
f    ::  Int -> Int -> Int
```

absolutely cannot do any I/O since `IO` does not appear in the returned type. This fact is often quite distressing to programmers used to placing print statements liberally throughout their code during debugging. There are, in fact, some unsafe functions available to get around this problem but these are better left to advanced programmers. Debugging packages (like `Trace`) often make liberal use of these 'forbidden functions' in an entirely safe manner.

7.2 Programming With Actions

I/O actions are ordinary Haskell values: they may be passed to functions, placed in structures, and used as any other Haskell value. Consider this list of actions:

```
todoList :: [IO ()]

todoList = [putChar 'a',
            do putChar 'b'
               putChar 'c',
            do c <- getChar
               putChar c]
```

This list doesn't actually invoke any actions—it simply holds them. To join these actions into a single action, a function such as `sequence_` is needed:

```
sequence_         :: [IO ()] -> IO ()
sequence_ []      =  return ()
sequence_ (a:as)  =  do a
                        sequence as
```

This can be simplified by noting that `do x;y` is expanded to `x >> y` (see Section 9.1). This pattern of recursion is captured by the `foldr` function (see the Prelude for a definition of `foldr`); a better definition of `sequence_` is:

```
sequence_         :: [IO ()] -> IO ()
sequence_         =  foldr (>>) (return ())
```

The `do` notation is a useful tool but in this case the underlying monadic operator, `>>`, is more appropriate. An understanding of the operators upon which `do` is built is quite useful to the Haskell programmer.

The `sequence_` function can be used to construct `putStr` from `putChar`:

```
putStr                  :: String -> IO ()
putStr s                =  sequence_ (map putChar s)
```

One of the differences between Haskell and conventional imperative programming can be seen in `putStr`. In an imperative language, mapping an imperative version of `putChar` over the string would be sufficient to print it. In Haskell, however, the `map` function does not perform any action. Instead it creates a list of actions, one for each character in the string. The folding operation in `sequence_` uses the `>>` function to combine all of the individual actions into a single action. The `return ()` used here is quite necessary – `foldr` needs a null action at the end of the chain of actions it creates (especially if there are no characters in the string!).

The Prelude and the libraries contains many functions which are useful for sequencing I/O actions. These are usually generalized to arbitrary monads; any function with a context including `Monad m =>` works with the `IO` type.

7.3 Exception Handling

So far, we have avoided the issue of exceptions during I/O operations. What would happen if `getChar` encounters an end of file?[13] To deal with exceptional conditions such as 'file not found'

[13]We use the term *error* for $\perp$: a condition which cannot be recovered from such as non-termination or pattern match failure. Exceptions, on the other hand, can be caught and handled within the I/O monad.

within the I/O monad, a handling mechanism is used, similar in functionality to the one in standard ML. No special syntax or semantics are used; exception handling is part of the definition of the I/O sequencing operations.

Errors are encoded using a special data type, `IOError`. This type represents all possible exceptions that may occur within the I/O monad. This is an abstract type: no constructors for `IOError` are available to the user. Predicates allow `IOError` values to be queried. For example, the function

```
isEOFError       :: IOError -> Bool
```

determines whether an error was caused by an end-of-file condition. By making `IOError` abstract, new sorts of errors may be added to the system without a noticeable change to the data type. The function `isEOFError` is defined in a separate library, `IO`, and must be explicitly imported into a program.

An *exception handler* has type `IOError -> IO a`. The `catch` function associates an exception handler with an action or set of actions:

```
catch                     :: IO a -> (IOError -> IO a) -> IO a
```

The arguments to `catch` are an action and a handler. If the action succeeds, its result is returned without invoking the handler. If an error occurs, it is passed to the handler as a value of type `IOError` and the action associated with the handler is then invoked. For example, this version of `getChar` returns a newline when an error is encountered:

```
getChar'                :: IO Char
getChar'                =  getChar `catch` (\e -> return '\n')
```

This is rather crude since it treats all errors in the same manner. If only end-of-file is to be recognized, the error value must be queried:

```
getChar'                :: IO Char
getChar'                =  getChar `catch` eofHandler where
    eofHandler e = if isEofError e then return '\n' else ioError e
```

The `ioError` function used here throws an exception on to the next exception handler. The type of `ioError` is

```
ioError                 :: IOError -> IO a
```

It is similar to `return` except that it transfers control to the exception handler instead of proceeding to the next I/O action. Nested calls to `catch` are permitted, and produce nested exception handlers.

Using `getChar'`, we can redefine `getLine` to demonstrate the use of nested handlers:

```
getLine'        :: IO String
getLine'        = catch getLine'' (\err -> return ("Error: " ++ show err))
        where
                   getLine'' = do c <- getChar'
                                  if c == '\n' then return ""
                                               else do l <- getLine'
                                                       return (c:l)
```

The nested error handlers allow `getChar'` to catch end of file while any other error results in a string starting with `"Error: "` from `getLine'`.

For convenience, Haskell provides a default exception handler at the topmost level of a program that prints out the exception and terminates the program.

7.4 Files, Channels, and Handles

Aside from the I/O monad and the exception handling mechanism it provides, I/O facilities in Haskell are for the most part quite similar to those in other languages. Many of these functions are in the `IO` library instead of the Prelude and thus must be explicitly imported to be in scope (modules and importing are discussed in Section 11). Also, many of these functions are discussed in the Library Report instead of the main report.

Opening a file creates a *handle* (of type `Handle`) for use in I/O transactions. Closing the handle closes the associated file:

```
type FilePath         =  String  -- path names in the file system
openFile              :: FilePath -> IOMode -> IO Handle
hClose                :: Handle -> IO ()
data IOMode           =  ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Handles can also be associated with *channels*: communication ports not directly attached to files. A few channel handles are predefined, including `stdin` (standard input), `stdout` (standard output), and `stderr` (standard error). Character level I/O operations include `hGetChar` and `hPutChar`, which take a handle as an argument. The `getChar` function used previously can be defined as:

```
getChar                = hGetChar stdin
```

Haskell also allows the entire contents of a file or channel to be returned as a single string:

```
getContents            :: Handle -> IO String
```

Pragmatically, it may seem that `getContents` must immediately read an entire file or channel, resulting in poor space and time performance under certain conditions. However, this is not the case. The key point is that `getContents` returns a "lazy" (i.e. non-strict) list of characters (recall that strings are just lists of characters in Haskell), whose elements are read "by demand" just like any other list. An implementation can be expected to implement this demand-driven behavior by reading one character at a time from the file as they are required by the computation.

In this example, a Haskell program copies one file to another:

```
main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
          toHandle   <- getAndOpenFile "Copy to: " WriteMode
          contents   <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          putStr "Done."

getAndOpenFile          :: String -> IOMode -> IO Handle

getAndOpenFile prompt mode =
    do putStr prompt
       name <- getLine
       catch (openFile name mode)
             (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
                       getAndOpenFile prompt mode)
```

By using the lazy `getContents` function, the entire contents of the file need not be read into memory all at once. If `hPutStr` chooses to buffer the output by writing the string in fixed sized blocks of characters, only one block of the input file needs to be in memory at once. The input file is closed implicitly when the last character has been read.

7.5 Haskell and Imperative Programming

As a final note, I/O programming raises an important issue: this style looks suspiciously like ordinary imperative programming. For example, the `getLine` function:

```
getLine         = do c <- getChar
                     if c == '\n'
                          then return ""
                          else do l <- getLine
                                  return (c:l)
```

bears a striking similarity to imperative code (not in any real language) :

```
function getLine() {
  c := getChar();
  if c == `\n` then return ""
               else {l := getLine();
                     return c:l}}
```

So, in the end, has Haskell simply re-invented the imperative wheel?

In some sense, yes. The I/O monad constitutes a small imperative sub-language inside Haskell, and thus the I/O component of a program may appear similar to ordinary imperative code. But there is one important difference: There is no special semantics that the user needs to deal with. In particular, equational reasoning in Haskell is not compromised. The imperative feel of the monadic code in a program does not detract from the functional aspect of Haskell. An experienced functional programmer should be able to minimize the imperative component of the program, only using

the I/O monad for a minimal amount of top-level sequencing. The monad cleanly separates the functional and imperative program components. In contrast, imperative languages with functional subsets do not generally have any well-defined barrier between the purely functional and imperative worlds.

8 Standard Haskell Classes

In this section we introduce the predefined standard type classes in Haskell. We have simplified these classes somewhat by omitting some of the less interesting methods in these classes; the Haskell report contains a more complete description. Also, some of the standard classes are part of the standard Haskell libraries; these are described in the Haskell Library Report.

8.1 Equality and Ordered Classes

The classes `Eq` and `Ord` have already been discussed. The definition of `Ord` in the Prelude is somewhat more complex than the simplified version of `Ord` presented earlier. In particular, note the `compare` method:

```
data Ordering           =  EQ | LT | GT
compare                 :: Ord a => a -> a -> Ordering
```

The `compare` method is sufficient to define all other methods (via defaults) in this class and is the best way to create `Ord` instances.

8.2 The Enumeration Class

Class `Enum` has a set of operations that underlie the syntactic sugar of arithmetic sequences; for example, the arithmetic sequence expression `[1,3..]` stands for `enumFromThen 1 3` (see §3.10 for the formal translation). We can now see that arithmetic sequence expressions can be used to generate lists of any type that is an instance of `Enum`. This includes not only most numeric types, but also `Char`, so that, for instance, `['a'..'z']` denotes the list of lower-case letters in alphabetical order. Furthermore, user-defined enumerated types like `Color` can easily be given `Enum` instance declarations. If so:

```
[Red .. Violet]   ⇒   [Red, Green, Blue, Indigo, Violet]
```

Note that such a sequence is *arithmetic* in the sense that the increment between values is constant, even though the values are not numbers. Most types in `Enum` can be mapped onto fixed precision integers; for these, the `fromEnum` and `toEnum` convert between `Int` and a type in `Enum`.

8.3 The Read and Show Classes

The instances of class `Show` are those types that can be converted to character strings (typically for I/O). The class `Read` provides operations for parsing character strings to obtain the values they may represent. The simplest function in the class `Show` is `show`:

```
show                    :: (Show a) => a -> String
```

Naturally enough, `show` takes any value of an appropriate type and returns its representation as a character string (list of characters), as in `show (2+2)`, which results in `"4"`. This is fine as far as it goes, but we typically need to produce more complex strings that may have the representations of many values in them, as in

```
"The sum of " ++ show x ++ " and " ++ show y ++ " is " ++ show (x+y) ++ "."
```

and after a while, all that concatenation gets to be a bit inefficient. Specifically, let's consider a function to represent the binary trees of Section 2.2.1 as a string, with suitable markings to show the nesting of subtrees and the separation of left and right branches (provided the element type is representable as a string):

```
showTree                :: (Show a) => Tree a -> String
showTree (Leaf x)       =  show x
showTree (Branch l r)   =  "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

Because `(++)` has time complexity linear in the length of its left argument, `showTree` is potentially quadratic in the size of the tree.

To restore linear complexity, the function `shows` is provided:

```
shows                   :: (Show a) => a -> String -> String
```

`shows` takes a printable value and a string and returns that string with the value's representation concatenated at the front. The second argument serves as a sort of string accumulator, and `show` can now be defined as `shows` with the null accumulator. This is the default definition of `show` in the `Show` class definition:

```
show x                  =  shows x ""
```

We can use `shows` to define a more efficient version of `showTree`, which also has a string accumulator argument:

```
showsTree               :: (Show a) => Tree a -> String -> String
showsTree (Leaf x) s    =  shows x s
showsTree (Branch l r) s=  '<' : showsTree l ('|' : showsTree r ('>' : s))
```

This solves our efficiency problem (`showsTree` has linear complexity), but the presentation of this function (and others like it) can be improved. First, let's create a type synonym:

```
type ShowS              =  String -> String
```

This is the type of a function that returns a string representation of something followed by an accumulator string. Second, we can avoid carrying accumulators around, and also avoid amassing parentheses at the right end of long constructions, by using functional composition:

```
showsTree               :: (Show a) => Tree a -> ShowS
showsTree (Leaf x)      =  shows x
showsTree (Branch l r)  =  ('<':) . showsTree l . ('|':) . showsTree r . ('>':)
```

Something more important than just tidying up the code has come about by this transformation: we have raised the presentation from an *object level* (in this case, strings) to a *function level*. We can think of the typing as saying that `showsTree` maps a tree into a *showing function*. Functions

like `('<' :)` or `("a string" ++)` are primitive showing functions, and we build up more complex functions by function composition.

Now that we can turn trees into strings, let's turn to the inverse problem. The basic idea is a parser for a type `a`, which is a function that takes a string and returns a list of `(a, String)` pairs [9]. The Prelude provides a type synonym for such functions:

```
type ReadS a            =  String -> [(a,String)]
```

Normally, a parser returns a singleton list, containing a value of type `a` that was read from the input string and the remaining string that follows what was parsed. If no parse was possible, however, the result is the empty list, and if there is more than one possible parse (an ambiguity), the resulting list contains more than one pair. The standard function `reads` is a parser for any instance of `Read`:

```
reads                   :: (Read a) => ReadS a
```

We can use this function to define a parsing function for the string representation of binary trees produced by `showsTree`. List comprehensions give us a convenient idiom for constructing such parsers:[14]

```
readsTree               :: (Read a) => ReadS (Tree a)
readsTree ('<':s)       =  [(Branch l r, u) | (l, '|':t) <- readsTree s,
                                              (r, '>':u) <- readsTree t ]
readsTree s             =  [(Leaf x, t)     | (x,t)      <- reads s]
```

Let's take a moment to examine this function definition in detail. There are two main cases to consider: If the first character of the string to be parsed is `'<'`, we should have the representation of a branch; otherwise, we have a leaf. In the first case, calling the rest of the input string following the opening angle bracket `s`, any possible parse must be a tree `Branch l r` with remaining string `u`, subject to the following conditions:

1. The tree `l` can be parsed from the beginning of the string `s`.
2. The string remaining (following the representation of `l`) begins with `'|'`. Call the tail of this string `t`.
3. The tree `r` can be parsed from the beginning of `t`.
4. The string remaining from that parse begins with `'>'`, and `u` is the tail.

Notice the expressive power we get from the combination of pattern matching with list comprehension: the form of a resulting parse is given by the main expression of the list comprehension, the first two conditions above are expressed by the first generator ("`(l, '|':t)` is drawn from the list of parses of `s`"), and the remaining conditions are expressed by the second generator.

The second defining equation above just says that to parse the representation of a leaf, we parse a representation of the element type of the tree and apply the constructor `Leaf` to the value thus obtained.

[14] An even more elegant approach to parsing uses monads and parser combinators. These are part of a standard parsing library distributed with most Haskell systems.

We'll accept on faith for the moment that there is a `Read` (and `Show`) instance of `Integer` (among many other types), providing a `reads` that behaves as one would expect, e.g.:

```
(reads "5 golden rings") :: [(Integer,String)]   ⇒   [(5, " golden rings")]
```

With this understanding, the reader should verify the following evaluations:

```
readsTree "<1|<2|3>>"    ⇒    [(Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)), "")]
readsTree "<1|2"         ⇒    []
```

There are a couple of shortcomings in our definition of `readsTree`. One is that the parser is quite rigid, allowing no white space before or between the elements of the tree representation; the other is that the way we parse our punctuation symbols is quite different from the way we parse leaf values and subtrees, this lack of uniformity making the function definition harder to read. We can address both of these problems by using the lexical analyzer provided by the Prelude:

```
lex                     :: ReadS String
```

`lex` normally returns a singleton list containing a pair of strings: the first lexeme in the input string and the remainder of the input. The lexical rules are those of Haskell programs, including comments, which `lex` skips, along with whitespace. If the input string is empty or contains only whitespace and comments, `lex` returns `[("","")]`; if the input is not empty in this sense, but also does not begin with a valid lexeme after any leading whitespace and comments, `lex` returns `[]`.

Using the lexical analyzer, our tree parser now looks like this:

```
readsTree               :: (Read a) => ReadS (Tree a)
readsTree s             =  [(Branch l r, x) | ("<", t) <- lex s,
                                              (l,   u) <- readsTree t,
                                              ("|", v) <- lex u,
                                              (r,   w) <- readsTree v,
                                              (">", x) <- lex w         ]
                           ++
                           [(Leaf x, t)     | (x,   t) <- reads s       ]
```

We may now wish to use `readsTree` and `showsTree` to declare `(Read a) => Tree a` an instance of `Read` and `(Show a) => Tree a` an instance of `Show`. This would allow us to use the generic overloaded functions from the Prelude to parse and display trees. Moreover, we would automatically then be able to parse and display many other types containing trees as components, for example, `[Tree Integer]`. As it turns out, `readsTree` and `showsTree` are of almost the right types to be `Show` and `Read` methods The `showsPrec` and `readsPrec` methods are parameterized versions of `shows` and `reads`. The extra parameter is a precedence level, used to properly parenthesize expressions containing infix constructors. For types such as `Tree`, the precedence can be ignored. The `Show` and `Read` instances for `Tree` are:

```
instance Show a => Show (Tree a) where
    showsPrec _ x = showsTree x

instance Read a => Read (Tree a) where
    readsPrec _ s = readsTree s
```

Alternatively, the `Show` instance could be defined in terms of `showTree`:

```
instance Show a => Show (Tree a) where
   show t = showTree t
```

This, however, will be less efficient than the `ShowS` version. Note that the `Show` class defines default methods for both `showsPrec` and `show`, allowing the user to define either one of these in an instance declaration. Since these defaults are mutually recursive, an instance declaration that defines neither of these functions will loop when called. Other classes such as `Num` also have these "interlocking defaults".

We refer the interested reader to §D for details of the `Read` and `Show` classes.

We can test the `Read` and `Show` instances by applying `(read . show)` (which should be the identity) to some trees, where `read` is a specialization of `reads`:

```
read                    :: (Read a) => String -> a
```

This function fails if there is not a unique parse or if the input contains anything more than a representation of one value of type `a` (and possibly, comments and whitespace).

8.4 Derived Instances

Recall the `Eq` instance for trees we presented in Section 5; such a declaration is simple—and boring—to produce: we require that the element type in the leaves be an equality type; then, two leaves are equal iff they contain equal elements, and two branches are equal iff their left and right subtrees are equal, respectively. Any other two trees are unequal:

```
instance  (Eq a) => Eq (Tree a)  where
    (Leaf x)     == (Leaf y)        =  x == y
    (Branch l r) == (Branch l' r')  =  l == l' && r == r'
    _            == _               =  False
```

Fortunately, we don't need to go through this tedium every time we need equality operators for a new type; the `Eq` instance can be *derived automatically* from the `data` declaration if we so specify:

```
data  Tree a            =  Leaf a | Branch (Tree a) (Tree a)  deriving Eq
```

The `deriving` clause implicitly produces an `Eq` instance declaration just like the one in Section 5. Instances of `Ord`, `Enum`, `Ix`, `Read`, and `Show` can also be generated by the `deriving` clause. [More than one class name can be specified, in which case the list of names must be parenthesized and the names separated by commas.]

The derived `Ord` instance for `Tree` is slightly more complicated than the `Eq` instance:

```
instance  (Ord a) => Ord (Tree a)  where
    (Leaf _)     <= (Branch _)      =  True
    (Leaf x)     <= (Leaf y)        =  x <= y
    (Branch _)   <= (Leaf _)        =  False
    (Branch l r) <= (Branch l' r')  =  l == l' && r <= r' || l <= l'
```

This specifies a *lexicographic* order: Constructors are ordered by the order of their appearance in

the `data` declaration, and the arguments of a constructor are compared from left to right. Recall that the built-in list type is semantically equivalent to an ordinary two-constructor type. In fact, this is the full declaration:

```
data [a]        = [] | a : [a] deriving (Eq, Ord)     -- pseudo-code
```

(Lists also have `Show` and `Read` instances, which are not derived.) The derived `Eq` and `Ord` instances for lists are the usual ones; in particular, character strings, as lists of characters, are ordered as determined by the underlying `Char` type, with an initial substring comparing less than a longer string; for example, `"cat" < "catalog"`.

In practice, `Eq` and `Ord` instances are almost always derived, rather than user-defined. In fact, we should provide our own definitions of equality and ordering predicates only with some trepidation, being careful to maintain the expected algebraic properties of equivalence relations and total orders. An intransitive `(==)` predicate, for example, could be disastrous, confusing readers of the program and confounding manual or automatic program transformations that rely on the `(==)` predicate's being an approximation to definitional equality. Nevertheless, it is sometimes necessary to provide `Eq` or `Ord` instances different from those that would be derived; probably the most important example is that of an abstract data type in which different concrete values may represent the same abstract value.

An enumerated type can have a derived `Enum` instance, and here again, the ordering is that of the constructors in the `data` declaration. For example:

```
data Day = Sunday | Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday         deriving (Enum)
```

Here are some simple examples using the derived instances for this type:

```
[Wednesday .. Friday]    ⇒   [Wednesday, Thursday, Friday]
[Monday, Wednesday ..]   ⇒   [Monday, Wednesday, Friday]
```

Derived `Read` (`Show`) instances are possible for all types whose component types also have `Read` (`Show`) instances. (`Read` and `Show` instances for most of the standard types are provided by the Prelude. Some types, such as the function type `(->)`, have a `Show` instance but not a corresponding `Read`.) The textual representation defined by a derived `Show` instance is consistent with the appearance of constant Haskell expressions of the type in question. For example, if we add `Show` and `Read` to the `deriving` clause for type `Day`, above, we obtain

```
show [Monday .. Wednesday]    ⇒   "[Monday,Tuesday,Wednesday]"
```

9 About Monads

Many newcomers to Haskell are puzzled by the concept of *monads*. Monads are frequently encountered in Haskell: the IO system is constructed using a monad, a special syntax for monads has been provided (`do` expressions), and the standard libraries contain an entire module dedicated to monads. In this section we explore monadic programming in more detail.

This section is perhaps less "gentle" than the others. Here we address not only the language features that involve monads but also try to reveal the bigger picture: why monads are such an important tool and how they are used. There is no single way of explaining monads that works for everyone; more explanations can be found at `haskell.org`. Another good introduction to practical programming using monads is Wadler's *Monads for Functional Programming* [10].

9.1 Monadic Classes

The Prelude contains a number of classes defining monads are they are used in Haskell. These classes are based on the monad construct in category theory; whilst the category theoretic terminology provides the names for the monadic classes and operations, it is not necessary to delve into abstract mathematics to get an intuitive understanding of how to use the monadic classes.

A monad is constructed on top of a polymorphic type such as `IO`. The monad itself is defined by instance declarations associating the type with the some or all of the monadic classes, `Functor`, `Monad`, and `MonadPlus`. None of the monadic classes are derivable. In addition to `IO`, two other types in the Prelude are members of the monadic classes: lists (`[]`) and `Maybe`.

Mathematically, monads are governed by set of *laws* that should hold for the monadic operations. This idea of laws is not unique to monads: Haskell includes other operations that are governed, at least informally, by laws. For example, `x /= y` and `not (x == y)` ought to be the same for any type of values being compared. However, there is no guarantee of this: both `==` and `/=` are separate methods in the `Eq` class and there is no way to assure that `==` and `=/` are related in this manner. In the same sense, the monadic laws presented here are not enforced by Haskell, but ought be obeyed by any instances of a monadic class. The monad laws give insight into the underlying structure of monads: by examining these laws, we hope to give a feel for how monads are used.

The `Functor` class, already discussed in section 5, defines a single operation: `fmap`. The map function applies an operation to the objects inside a container (polymorphic types can be thought of as containers for values of another type), returning a container of the same shape. These laws apply to `fmap` in the class `Functor`:

```
fmap id       =  id
fmap (f . g)  =  fmap f . fmap g
```

These laws ensure that the container shape is unchanged by `fmap` and that the contents of the container are not re-arranged by the mapping operation.

The `Monad` class defines two basic operators: `>>=` (bind) and `return`.

```
infixl 1  >>, >>=
class  Monad m  where
    (>>=)            :: m a -> (a -> m b) -> m b
    (>>)             :: m a -> m b -> m b
    return           :: a -> m a
    fail             :: String -> m a

    m >> k           =  m >>= \_ -> k
```

The bind operations, `>>` and `>>=`, combine two monadic values while the `return` operation injects

a value into the monad (container). The signature of `>>=` helps us to understand this operation: `ma >>= \v -> mb` combines a monadic value `ma` containing values of type `a` and a function which operates on a value `v` of type `a`, returning the monadic value `mb`. The result is to combine `ma` and `mb` into a monadic value containing `b`. The `>>` function is used when the function does not need the value produced by the first monadic operator.

The precise meaning of binding depends, of course, on the monad. For example, in the IO monad, `x >>= y` performs two actions sequentially, passing the result of the first into the second. For the other built-in monads, lists and the `Maybe` type, these monadic operations can be understood in terms of passing zero or more values from one calculation to the next. We will see examples of this shortly.

The `do` syntax provides a simple shorthand for chains of monadic operations. The essential translation of `do` is captured in the following two rules:

```
do e1 ; e2      =       e1 >> e2
do p <- e1; e2  =       e1 >>= \p -> e2
```

When the pattern in this second form of `do` is refutable, pattern match failure calls the `fail` operation. This may raise an error (as in the `IO` monad) or return a "zero" (as in the list monad). Thus the more complex translation is

```
do p <- e1; e2  =   e1 >>= (\v -> case v of p -> e2; _ -> fail "s")
```

where `"s"` is a string identifying the location of the `do` statement for possible use in an error message. For example, in the I/O monad, an action such as `'a' <- getChar` will call `fail` if the character typed is not 'a'. This, in turn, terminates the program since in the I/O monad `fail` calls `error`.

The laws which govern `>>=` and `return` are:

```
return a >>= k          =  k a
m >>= return            =  m
xs >>= return . f       =  fmap f xs
m >>= (\x -> k x >>= h) =  (m >>= k) >>= h
```

The class `MonadPlus` is used for monads that have a *zero* element and a *plus* operation:

```
class  (Monad m) => MonadPlus m  where
    mzero             :: m a
    mplus             :: m a -> m a -> m a
```

The zero element obeys the following laws:

```
m >>= \x -> mzero  =  mzero
mzero >>= m        =  mzero
```

For lists, the zero value is `[]`, the empty list. The I/O monad has no zero element and is not a member of this class.

The laws governing the `mplus` operator are as follows:

```
m `mplus` mzero  =  m
mzero `mplus` m  =  m
```

The `mplus` operator is ordinary list concatenation in the list monad.

9.2 Built-in Monads

Given the monadic operations and the laws that govern them, what can we build? We have already examined the I/O monad in detail so we start with the two other built-in monads.

For lists, monadic binding involves joining together a set of calculations for each value in the list. When used with lists, the signature of `>>=` becomes:

```
(>>=)                    :: [a] -> (a -> [b]) -> [b]
```

That is, given a list of `a`'s and a function that maps an `a` onto a list of `b`'s, binding applies this function to each of the `a`'s in the input and returns all of the generated `b`'s concatenated into a list. The `return` function creates a singleton list. These operations should already be familiar: list comprehensions can easily be expressed using the monadic operations defined for lists. These following three expressions are all different syntax for the same thing:

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]
```

```
do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)
```

```
[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
   (\r -> case r of True -> return (x,y)
                    _    -> fail "")))
```

This definition depends on the definition of `fail` in this monad as the empty list. Essentially, each `<-` is generating a set of values which is passed on into the remainder of the monadic computation. Thus `x <- [1,2,3]` invokes the remainder of the monadic computation three times, once for each element of the list. The returned expression, `(x,y)`, will be evaluated for all possible combinations of bindings that surround it. In this sense, the list monad can be thought of as describing functions of multi-valued arguments. For example, this function:

```
mvLift2                 :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y           =  do x' <- x
                              y' <- y
                              return (f x' y')
```

turns an ordinary function of two arguments (`f`) into a function over multiple values (lists of arguments), returning a value for each possible combination of the two input arguments. For example,

```
mvLift2 (+) [1,3] [10,20,30]       ⇒    [11,21,31,13,23,33]
mvLift2 (\a b->[a,b]) "ab" "cd"    ⇒    ["ac","ad","bc","bd"]
mvLift2 (*) [1,2,4] []             ⇒    []
```

This function is a specialized version of the `LiftM2` function in the monad library. You can think of it as transporting a function from outside the list monad, `f`, into the list monad in which computations take on multiple values.

The monad defined for `Maybe` is similar to the list monad: the value `Nothing` serves as `[]` and `Just x` as `[x]`.

9.3 Using Monads

Explaining the monadic operators and their associated laws doesn't really show what monads are good for. What they really provide is *modularity*. That is, by defining an operation monadically, we can hide underlying machinery in a way that allows new features to be incorporated into the monad transparently. Wadler's paper [10] is an excellent example of how monads can be used to construct modular programs. We will start with a monad taken directly from this paper, the state monad, and then build a more complex monad with a similar definition.

Briefly, a state monad built around a state type `S` looks like this:

```
data SM a = SM (S -> (a,S))  -- The monadic type

instance Monad SM where
  -- defines state propagation
  SM c1 >>= fc2         =  SM (\s0 -> let (r,s1) = c1 s0
                                          SM c2 = fc2 r in
                                      c2 s1)
  return k              =  SM (\s -> (k,s))

 -- extracts the state from the monad
readSM                  :: SM S
readSM                  =  SM (\s -> (s,s))

 -- updates the state of the monad
updateSM                :: (S -> S) -> SM ()  -- alters the state
updateSM f              =  SM (\s -> ((), f s))

-- run a computation in the SM monad
runSM                   :: S -> SM a -> (a,S)
runSM s0 (SM c)         =  c s0
```

This example defines a new type, `SM`, to be a computation that implicitly carries a type `S`. That is, a computation of type `SM t` defines a value of type `t` while also interacting with (reading and writing) the state of type `S`. The definition of `SM` is simple: it consists of functions that take a state and produce two results: a returned value (of any type) and an updated state. We can't use a type synonym here: we need a type name like `SM` that can be used in instance declarations. The `newtype` declaration is often used here instead of `data`.

This instance declaration defines the 'plumbing' of the monad: how to sequence two computations and the definition of an empty computation. Sequencing (the `>>=` operator) defines a computation (denoted by the constructor `SM`) that passes an initial state, `s0`, into `c1`, then passes the value coming out of this computation, `r`, to the function that returns the second computation, `c2`. Finally, the state coming out of `c1` is passed into `c2` and the overall result is the result of `c2`.

The definition of `return` is easier: `return` doesn't change the state at all; it only serves to bring a value into the monad.

While `>>=` and `return` are the basic monadic sequencing operations, we also need some *monadic primitives*. A monadic primitive is simply an operation that uses the insides of the monad abstraction and taps into the 'wheels and gears' that make the monad work. For example, in the `IO` monad,

operators such as `putChar` are primitive since they deal with the inner workings of the `IO` monad. Similarly, our state monad uses two primitives: `readSM` and `updateSM`. Note that these depend on the inner structure of the monad - a change to the definition of the `SM` type would require a change to these primitives.

The definition of `readSM` and `updateSM` are simple: `readSM` brings the state out of the monad for observation while `updateSM` allows the user to alter the state in the monad. (We could also have used `writeSM` as a primitive but update is often a more natural way of dealing with state).

Finally, we need a function that runs computations in the monad, `runSM`. This takes an initial state and a computation and yields both the returned value of the computation and the final state.

Looking at the bigger picture, what we are trying to do is define an overall computation as a series of steps (functions with type `SM a`), sequenced using `>>=` and `return`. These steps may interact with the state (via `readSM` or `updateSM`) or may ignore the state. However, the use (or non-use) of the state is hidden: we don't invoke or sequence our computations differently depending on whether or not they use `S`.

Rather than present any examples using this simple state monad, we proceed on to a more complex example that includes the state monad. We define a small *embedded language* of resource-using calculations. That is, we build a special purpose language implemented as a set of Haskell types and functions. Such languages use the basic tools of Haskell, functions and types, to build a library of operations and types specifically tailored to a domain of interest.

In this example, consider a computation that requires some sort of resource. If the resource is available, computation proceeds; when the resource is unavailable, the computation suspends. We use the type `R` to denote a computation using resources controlled by our monad. The definition of `R` is as follows:

```
data R a = R (Resource -> (Resource, Either a (R a)))
```

Each computation is a function from available resources to remaining resources, coupled with either a result, of type `a`, or a suspended computation, of type `R a`, capturing the work done up to the point where resources were exhausted.

The `Monad` instance for `R` is as follows:

```
instance Monad R where
  R c1 >>= fc2          = R (\r -> case c1 r of
                                (r', Left v)    -> let R c2 = fc2 v in
                                                   c2 r'
                                (r', Right pc1) -> (r', Right (pc1 >>= fc2)))
  return v              = R (\r -> (r, (Left v)))
```

The `Resource` type is used in the same manner as the state in the state monad. This definition reads as follows: to combine two 'resourceful' computations, `c1` and `fc2` (a function producing `c2`), pass the initial resources into `c1`. The result will be either

- a value, `v`, and remaining resources, which are used to determine the next computation (the call `fc2 v`), or
- a suspended computation, `pc1`, and resources remaining at the point of suspension.

The suspension must take the second computation into consideration: `pc1` suspends only the first computation, `c1`, so we must bind `c2` to this to produce a suspension of the overall computation. The definition of `return` leaves the resources unchanged while moving `v` into the monad.

This instance declaration defines the basic structure of the monad but does not determine how resources are used. This monad could be used to control many types of resource or implement many different types of resource usage policies. We will demonstrate a very simple definition of resources as an example: we choose `Resource` to be an `Integer`, representing available computation steps:

```
type Resource           =  Integer
```

This function takes a step unless no steps are available:

```
step                    :: a -> R a
step v                  =  c where
                              c = R (\r -> if r /= 0 then (r-1, Left v)
                                                     else (r, Right c))
```

The `Left` and `Right` constructors are part of the `Either` type. This function continues computation in `R` by returning `v` so long as there is at least one computational step resource available. If no steps are available, the `step` function suspends the current computation (this suspension is captured in `c`) and passes this suspended computation back into the monad.

So far, we have the tools to define a sequence of "resourceful" computations (the monad) and we can express a form of resource usage using `step`. Finally, we need to address how computations in this monad are expressed.

Consider an increment function in our monad:

```
inc                     :: R Integer -> R Integer
inc i                   =  do iValue <- i
                              step (iValue+1)
```

This defines increment as a single step of computation. The `<-` is necessary to pull the argument value out of the monad; the type of `iValue` is `Integer` instead of `R Integer`.

This definition isn't particularly satisfying, though, compared to the standard definition of the increment function. Can we instead "dress up" existing operations like `+` so that they work in our monadic world? We'll start with a set of `lifting` functions. These bring existing functionality into the monad. Consider the definition of `lift1` (this is slightly different from the `liftM1` found in the `Monad` library):

```
lift1                   :: (a -> b) -> (R a -> R b)
lift1 f                 =  \ra1 -> do a1 <- ra1
                                      step (f a1)
```

This takes a function of a single argument, `f`, and creates a function in `R` that executes the lifted function in a single step. Using `lift1`, `inc` becomes

```
inc                     :: R Integer -> R Integer
inc i                   =  lift1 (i+1)
```

This is better but still not ideal. First, we add `lift2`:

```
lift2                   :: (a -> b -> c) -> (R a -> R b -> R c)
lift2 f                 =  \ra1 ra2 -> do a1 <- ra1
                                          a2 <- ra2
                                          step (f a1 a2)
```

Notice that this function explicitly sets the order of evaluation in the lifted function: the computation yielding `a1` occurs before the computation for `a2`.

Using `lift2`, we can create a new version of `==` in the `R` monad:

```
(==*)                   :: Ord a => R a -> R a -> R Bool
(==*)                   =  lift2 (==)
```

We had to use a slightly different name for this new function since `==` is already taken but in some cases we can use the same name for the lifted and unlifted function. This instance declaration allows all of the operators in `Num` to be used in `R`:

```
instance Num a => Num (R a) where
  (+)                   =  lift2 (+)
  (-)                   =  lift2 (-)
  negate                =  lift1 negate
  (*)                   =  lift2 (*)
  abs                   =  lift1 abs
  fromInteger           =  return . fromInteger
```

The `fromInteger` function is applied implicitly to all integer constants in a Haskell program (see Section 10.3); this definition allows integer constants to have the type `R Integer`. We can now, finally, write increment in a completely natural style:

```
inc                     :: R Integer -> R Integer
inc x                   =  x + 1
```

Note that we cannot lift the `Eq` class in the same manner as the `Num` class: the signature of `==*` is not compatible with allowable overloadings of `==` since the result of `==*` is `R Bool` instead of `Bool`.

To express interesting computations in `R` we will need a conditional. Since we can't use `if` (it requires that the test be of type `Bool` instead of `R Bool`), we name the function `ifR`:

```
ifR                     :: R Bool -> R a -> R a -> R a
ifR tst thn els         =  do t <- tst
                              if t then thn else els
```

Now we're ready for a larger program in the `R` monad:

```
fact                    :: R Integer -> R Integer
fact x                  =  ifR (x ==* 0) 1 (x * fact (x-1))
```

Now this isn't quite the same as an ordinary factorial function but still quite readable. The idea of providing new definitions for existing operations like `+` or `if` is an essential part of creating an embedded language in Haskell. Monads are particularly useful for encapsulating the semantics of these embedded languages in a clean and modular way.

We're now ready to actually run some programs. This function runs a program in `M` given a maximum number of computation steps:

```
run                     :: Resource -> R a -> Maybe a
run s (R p)             =  case (p s) of
                             (_, Left v) -> Just v
                             _           -> Nothing
```

We use the `Maybe` type to deal with the possibility of the computation not finishing in the allotted number of steps. We can now compute

```
run 10 (fact 2)     ⇒    Just 2
run 10 (fact 20)    ⇒    Nothing
```

Finally, we can add some more interesting functionality to this monad. Consider the following function:

```
(|||)                   :: R a -> R a -> R a
```

This runs two computations in parallel, returning the value of the first one to complete. One possible definition of this function is:

```
c1 ||| c2               =  oneStep c1 (\c1' -> c2 ||| c1')
   where
        oneStep         :: R a -> (R a -> R a) -> R a
        oneStep (R c1) f =
             R (\r -> case c1 1 of
                        (r', Left v) -> (r+r'-1, Left v)
                        (r', Right c1') ->  -- r' must be 0
                         let R next = f c1' in
                           next (r+r'-1))
```

This takes a step in `c1`, returning its value of `c1` complete or, if `c1` returns a suspended computation (`c1'`), it evaluates `c2 ||| c1'`. The `oneStep` function takes a single step in its argument, either returning an evaluated value or passing the remainder of the computation into `f`. The definition of `oneStep` is simple: it gives `c1` a 1 as its resource argument. If a final value is reached, this is returned, adjusting the returned step count (it is possible that a computation might return after taking no steps so the returned resource count isn't necessarily 0). If the computation suspends, a patched up resource count is passed to the final continuation.

We can now evaluate expressions like `run 100 (fact (-1) ||| (fact 3))` without looping since the two calculations are interleaved. (Our definition of `fact` loops for `-1`). Many variations are possible on this basic structure. For example, we could extend the state to include a trace of the computation steps. We could also embed this monad inside the standard `IO` monad, allowing computations in `M` to interact with the outside world.

While this example is perhaps more advanced than others in this tutorial, it serves to illustrate the power of monads as a tool for defining the basic semantics of a system. We also present this example as a model of a small *Domain Specific Language*, something Haskell is particularly good at defining. Many other DSLs have been developed in Haskell; see `haskell.org` for many more examples. Of particular interest are Fran, a language of reactive animations, and Haskore, a language of computer music.

10 Numbers

Haskell provides a rich collection of numeric types, based on those of Scheme [7], which in turn are based on Common Lisp [8]. (Those languages, however, are dynamically typed.) The standard types include fixed- and arbitrary-precision integers, ratios (rational numbers) formed from each integer type, and single- and double-precision real and complex floating-point. We outline here the basic characteristics of the numeric type class structure and refer the reader to §6.4 for details.

10.1 Numeric Class Structure

The numeric type classes (class `Num` and those that lie below it) account for many of the standard Haskell classes. We also note that `Num` is a subclass of `Eq`, but not of `Ord`; this is because the order predicates do not apply to complex numbers. The subclass `Real` of `Num`, however, is a subclass of `Ord` as well.

The `Num` class provides several basic operations common to all numeric types; these include, among others, addition, subtraction, negation, multiplication, and absolute value:

```
(+), (-), (*)           :: (Num a) => a -> a -> a
negate, abs             :: (Num a) => a -> a
```

[`negate` is the function applied by Haskell's only prefix operator, minus; we can't call it `(-)`, because that is the subtraction function, so this name is provided instead. For example, `-x*y` is equivalent to `negate (x*y)`. (Prefix minus has the same syntactic precedence as infix minus, which, of course, is lower than that of multiplication.)]

Note that `Num` does *not* provide a division operator; two different kinds of division operators are provided in two non-overlapping subclasses of `Num`:

The class `Integral` provides whole-number division and remainder operations. The standard instances of `Integral` are `Integer` (unbounded or mathematical integers, also known as "bignums") and `Int` (bounded, machine integers, with a range equivalent to at least 29-bit signed binary). A particular Haskell implementation might provide other integral types in addition to these. Note that `Integral` is a subclass of `Real`, rather than of `Num` directly; this means that there is no attempt to provide Gaussian integers.

All other numeric types fall in the class `Fractional`, which provides the ordinary division operator `(/)`. The further subclass `Floating` contains trigonometric, logarithmic, and exponential functions.

The `RealFrac` subclass of `Fractional` and `Real` provides a function `properFraction`, which decomposes a number into its whole and fractional parts, and a collection of functions that round to integral values by differing rules:

```
properFraction          :: (Fractional a, Integral b) => a -> (b,a)
truncate, round,
floor, ceiling:         :: (Fractional a, Integral b) => a -> b
```

The `RealFloat` subclass of `Floating` and `RealFrac` provides some specialized functions for efficient access to the components of a floating-point number, the *exponent* and *significand*. The standard types `Float` and `Double` fall in class `RealFloat`.

10.2 Constructed Numbers

Of the standard numeric types, `Int`, `Integer`, `Float`, and `Double` are primitive. The others are made from these by type constructors.

`Complex` (found in the library `Complex`) is a type constructor that makes a complex type in class `Floating` from a `RealFloat` type:

```
data (RealFloat a) => Complex a = !a :+ !a  deriving (Eq, Text)
```

The `!` symbols are strictness flags; these were discussed in Section 6.3. Notice the context `RealFloat a`, which restricts the argument type; thus, the standard complex types are `Complex Float` and `Complex Double`. We can also see from the `data` declaration that a complex number is written x `:+` y; the arguments are the cartesian real and imaginary parts, respectively. Since `:+` is a data constructor, we can use it in pattern matching:

```
conjugate               :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y)        =  x :+ (-y)
```

Similarly, the type constructor `Ratio` (found in the `Rational` library) makes a rational type in class `RealFrac` from an instance of `Integral`. (`Rational` is a type synonym for `Ratio Integer`.) `Ratio`, however, is an abstract type constructor. Instead of a data constructor like `:+`, rationals use the '`%`' function to form a ratio from two integers. Instead of pattern matching, component extraction functions are provided:

```
(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
```

Why the difference? Complex numbers in cartesian form are unique—there are no nontrivial identities involving `:+`. On the other hand, ratios are not unique, but have a canonical (reduced) form that the implementation of the abstract data type must maintain; it is not necessarily the case, for instance, that `numerator (x%y)` is equal to `x`, although the real part of `x:+y` is always `x`.

10.3 Numeric Coercions and Overloaded Literals

The Standard Prelude and libraries provide several overloaded functions that serve as explicit coercions:

```
fromInteger             :: (Num a) => Integer -> a
fromRational            :: (Fractional a) => Rational -> a
toInteger               :: (Integral a) => a -> Integer
toRational              :: (RealFrac a) => a -> Rational
fromIntegral            :: (Integral a, Num b) => a -> b
fromRealFrac            :: (RealFrac a, Fractional b) => a -> b

fromIntegral            =  fromInteger . toInteger
fromRealFrac            =  fromRational . toRational
```

Two of these are implicitly used to provide overloaded numeric literals: An integer numeral (without a decimal point) is actually equivalent to an application of `fromInteger` to the value of the numeral as an `Integer`. Similarly, a floating numeral (with a decimal point) is regarded as an application of `fromRational` to the value of the numeral as a `Rational`. Thus, `7` has the type `(Num a) => a`, and `7.3` has the type `(Fractional a) => a`. This means that we can use numeric literals in generic numeric functions, for example:

```
halve                   :: (Fractional a) => a -> a
halve x                 =  x * 0.5
```

This rather indirect way of overloading numerals has the additional advantage that the method of interpreting a numeral as a number of a given type can be specified in an `Integral` or `Fractional` instance declaration (since `fromInteger` and `fromRational` are operators of those classes, respectively). For example, the `Num` instance of `(RealFloat a) => Complex a` contains this method:

```
fromInteger x           =  fromInteger x :+ 0
```

This says that a `Complex` instance of `fromInteger` is defined to produce a complex number whose real part is supplied by an appropriate `RealFloat` instance of `fromInteger`. In this manner, even user-defined numeric types (say, quaternions) can make use of overloaded numerals.

As another example, recall our first definition of `inc` from Section 2:

```
inc                     :: Integer -> Integer
inc n                   =  n+1
```

Ignoring the type signature, the most general type of `inc` is `(Num a) => a->a`. The explicit type signature is legal, however, since it is *more specific* than the principal type (a more general type signature would cause a static error). The type signature has the effect of restricting `inc`'s type, and in this case would cause something like `inc (1::Float)` to be ill-typed.

10.4 Default Numeric Types

Consider the following function definition:

```
rms             :: (Floating a) => a -> a -> a
rms x y         =  sqrt ((x^2 + y^2) * 0.5)
```

The exponentiation function `(^)` (one of three different standard exponentiation operators with different typings, see §6.8.5) has the type `(Num a, Integral b) => a -> b -> a`, and since `2` has the type `(Num a) => a`, the type of `x^2` is `(Num a, Integral b) => a`. This is a problem; there

is no way to resolve the overloading associated with the type variable `b`, since it is in the context, but has otherwise vanished from the type expression. Essentially, the programmer has specified that `x` should be squared, but has not specified whether it should be squared with an `Int` or an `Integer` value of two. Of course, we can fix this:

```
rms x y          =  sqrt ((x ^ (2::Integer) + y ^ (2::Integer)) * 0.5)
```

It's obvious that this sort of thing will soon grow tiresome, however.

In fact, this kind of overloading ambiguity is not restricted to numbers:

```
show (read "xyz")
```

As what type is the string supposed to be read? This is more serious than the exponentiation ambiguity, because there, any `Integral` instance will do, whereas here, very different behavior can be expected depending on what instance of `Text` is used to resolve the ambiguity.

Because of the difference between the numeric and general cases of the overloading ambiguity problem, Haskell provides a solution that is restricted to numbers: Each module may contain a *default declaration,* consisting of the keyword `default` followed by a parenthesized, comma-separated list of numeric monotypes (types with no variables). When an ambiguous type variable is discovered (such as `b`, above), if at least one of its classes is numeric and all of its classes are standard, the default list is consulted, and the first type from the list that will satisfy the context of the type variable is used. For example, if the default declaration `default (Int, Float)` is in effect, the ambiguous exponent above will be resolved as type `Int`. (See §4.3.4 for more details.)

The "default default" is `(Integer, Double)`, but `(Integer, Rational, Double)` may also be appropriate. Very cautious programmers may prefer `default ()`, which provides no defaults.

11 Modules

A Haskell program consists of a collection of *modules.* A module in Haskell serves the dual purpose of controlling name-spaces and creating abstract data types.

The top level of a module contains any of the various declarations we have discussed: fixity declarations, data and type declarations, class and instance declarations, type signatures, function definitions, and pattern bindings. Except for the fact that import declarations (to be described shortly) must appear first, the declarations may appear in any order (the top-level scope is mutually recursive).

Haskell's module design is relatively conservative: the name-space of modules is completely flat, and modules are in no way "first-class." Module names are alphanumeric and must begin with an uppercase letter. There is no formal connection between a Haskell module and the file system that would (typically) support it. In particular, there is no connection between module names and file names, and more than one module could conceivably reside in a single file (one module may even span several files). Of course, a particular implementation will most likely adopt conventions that make the connection between modules and files more stringent.

Technically speaking, a module is really just one big declaration which begins with the keyword `module`; here's an example for a module whose name is `Tree`:

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a                 = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)             = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

The type `Tree` and the function `fringe` should be familiar; they were given as examples in Section 2.2.1. [Because of the `where` keyword, layout is active at the top level of a module, and thus the declarations must all line up in the same column (typically the first). Also note that the module name is the same as that of the type; this is allowed.]

This module explicitly *exports* `Tree`, `Leaf`, `Branch`, and `fringe`. If the export list following the `module` keyword is omitted, *all* of the names bound at the top level of the module would be exported. (In the above example everything is explicitly exported, so the effect would be the same.) Note that the name of a type and its constructors have be grouped together, as in `Tree(Leaf,Branch)`. As short-hand, we could also write `Tree(..)`. Exporting a subset of the constructors is also possible. The names in an export list need not be local to the exporting module; any name in scope may be listed in an export list.

The `Tree` module may now be *imported* into some other module:

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

The various items being imported into and exported out of a module are called *entities*. Note the explicit import list in the import declaration; omitting it would cause all entities exported from `Tree` to be imported.

11.1 Qualified Names

There is an obvious problem with importing names directly into the namespace of module. What if two imported modules contain different entities with the same name? Haskell solves this problem using *qualified names*. An import declaration may use the `qualified` keyword to cause the imported names to be prefixed by the name of the module imported. These prefixes are followed by the '.' character without intervening whitespace. [Qualifiers are part of the lexical syntax. Thus, `A.x` and `A . x` are quite different: the first is a qualified name and the second a use of the infix '.' function.] For example, using the `Tree` module introduced above:

```
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]   -- A different definition of fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x

module Main where
import Tree ( Tree(Leaf,Branch), fringe )
import qualified Fringe ( fringe )

main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

Some Haskell programmers prefer to use qualifiers for all imported entities, making the source of each name explicit with every use. Others prefer short names and only use qualifiers when absolutely necessary.

Qualifiers are used to resolve conflicts between different entities which have the same name. But what if the same entity is imported from more than one module? Fortunately, such name clashes are allowed: an entity can be imported by various routes without conflict. The compiler knows whether entities from different modules are actually the same.

11.2 Abstract Data Types

Aside from controlling namespaces, modules provide the only way to build abstract data types (ADTs) in Haskell. For example, the characteristic feature of an ADT is that the *representation type* is *hidden*; all operations on the ADT are done at an abstract level which does not depend on the representation. For example, although the `Tree` type is simple enough that we might not normally make it abstract, a suitable ADT for it might include the following operations:

```
data Tree a             -- just the type name
leaf                    :: a -> Tree a
branch                  :: Tree a -> Tree a -> Tree a
cell                    :: Tree a -> a
left, right             :: Tree a -> Tree a
isLeaf                  :: Tree a -> Bool
```

A module supporting this is:

```
module TreeADT (Tree, leaf, branch, cell,
                left, right, isLeaf) where

data Tree a             = Leaf a | Branch (Tree a) (Tree a)

leaf                    = Leaf
branch                  = Branch
cell  (Leaf a)          = a
left  (Branch l r)      = l
right (Branch l r)      = r
isLeaf   (Leaf _)       = True
isLeaf   _              = False
```

Note in the export list that the type name `Tree` appears alone (i.e. without its constructors). Thus `Leaf` and `Branch` are not exported, and the only way to build or take apart trees outside of the module is by using the various (abstract) operations. Of course, the advantage of this information hiding is that at a later time we could *change* the representation type without affecting users of the type.

11.3 More Features

Here is a brief overview of some other aspects of the module system. See the report for more details.

- An `import` declaration may selectively hide entities using a `hiding` clause in the import declaration. This is useful for explicitly excluding names that are used for other purposes without having to use qualifiers for other imported names from the module.
- An `import` may contain an `as` clause to specify a different qualifier than the name of the importing module. This can be used to shorten qualifiers from modules with long names or to easily adapt to a change in module name without changing all qualifiers.
- Programs implicitly import the `Prelude` module. An explicit import of the Prelude overrides the implicit import of all Prelude names. Thus,

  ```
  import Prelude hiding length
  ```

  will not import `length` from the Standard Prelude, allowing the name `length` to be defined differently.
- Instance declarations are not explicitly named in import or export lists. Every module exports all of its instance declarations and every import brings all instance declarations into scope.
- Class methods may be named either in the manner of data constructors, in parentheses following the class name, or as ordinary variables.

Although Haskell's module system is relatively conservative, there are many rules concerning the import and export of values. Most of these are obvious—for instance, it is illegal to import two different entities having the same name into the same scope. Other rules are not so obvious—for example, for a given type and class, there cannot be more than one `instance` declaration for that combination of type and class anywhere in the program. The reader should read the Report for details (§5).

12 Typing Pitfalls

This short section give an intuitive description of a few common problems that novices run into using Haskell's type system.

12.1 Let-Bound Polymorphism

Any language using the Hindley-Milner type system has what is called *let-bound polymorphism*, because identifiers not bound using a `let` or `where` clause (or at the top level of a module) are limited with respect to their polymorphism. In particular, a *lambda-bound* function (i.e., one passed as argument to another function) cannot be instantiated in two different ways. For example, this program is illegal:

```
let f g  =  (g [], g 'a')                    -- ill-typed expression
in f (\x->x)
```

because `g`, bound to a lambda abstraction whose principal type is `a->a`, is used within `f` in two different ways: once with type `[a]->[a]`, and once with type `Char->Char`.

12.2 Numeric Overloading

It is easy to forget at times that numerals are *overloaded,* and *not implicitly coerced* to the various numeric types, as in many other languages. More general numeric expressions sometimes cannot be quite so generic. A common numeric typing error is something like the following:

```
average xs              =  sum xs / length xs          -- Wrong!
```

`(/)` requires fractional arguments, but `length`'s result is an `Int`. The type mismatch must be corrected with an explicit coercion:

```
average                 :: (Fractional a) => [a] -> a
average xs              =  sum xs / fromIntegral (length xs)
```

12.3 The Monomorphism Restriction

The Haskell type system contains a restriction related to type classes that is not found in ordinary Hindley-Milner type systems: the *monomorphism restriction.* The reason for this restriction is related to a subtle type ambiguity and is explained in full detail in the Report (§4.5.5). A simpler explanation follows:

The monomorphism restriction says that any identifier bound by a pattern binding (which includes bindings to a single identifier), and having no explicit type signature, must be *monomorphic.* An identifier is monomorphic if is either not overloaded, or is overloaded but is used in at most one specific overloading and is not exported.

Violations of this restriction result in a static type error. The simplest way to avoid the problem is to provide an explicit type signature. Note that *any* type signature will do (as long it is type correct).

A common violation of the restriction happens with functions defined in a higher-order manner, as in this definition of `sum` from the Standard Prelude:

```
sum                     =  foldl (+) 0
```

As is, this would cause a static type error. We can fix the problem by adding the type signature:

```
sum                     :: (Num a) => [a] -> a
```

Also note that this problem would not have arisen if we had written:

```
sum xs                  =  foldl (+) 0 xs
```

because the restriction only applies to pattern bindings.

13 Arrays

Ideally, arrays in a functional language would be regarded simply as functions from indices to values, but pragmatically, in order to assure efficient access to array elements, we need to be sure we can take advantage of the special properties of the domains of these functions, which are isomorphic to finite contiguous subsets of the integers. Haskell, therefore, does not treat arrays as general functions with an application operation, but as abstract data types with a subscript operation.

Two main approaches to functional arrays may be discerned: *incremental* and *monolithic* definition. In the incremental case, we have a function that produces an empty array of a given size and another that takes an array, an index, and a value, producing a new array that differs from the old one only at the given index. Obviously, a naive implementation of such an array semantics would be intolerably inefficient, either requiring a new copy of an array for each incremental redefinition, or taking linear time for array lookup; thus, serious attempts at using this approach employ sophisticated static analysis and clever run-time devices to avoid excessive copying. The monolithic approach, on the other hand, constructs an array all at once, without reference to intermediate array values. Although Haskell has an incremental array update operator, the main thrust of the array facility is monolithic.

Arrays are not part of the Standard Prelude—the standard library contains the array operators. Any module using arrays must import the `Array` module.

13.1 Index types

The `Ix` library defines a type class of array indices:

```
class  (Ord a) => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) a -> Int
    inRange     :: (a,a) -> a -> Bool
```

Instance declarations are provided for `Int`, `Integer`, `Char`, `Bool`, and tuples of `Ix` types up to length 5; in addition, instances may be automatically derived for enumerated and tuple types. We regard the primitive types as vector indices, and tuples as indices of multidimensional rectangular arrays. Note that the first argument of each of the operations of class `Ix` is a pair of indices; these are typically the *bounds* (first and last indices) of an array. For example, the bounds of a 10-element, zero-origin vector with `Int` indices would be `(0,9)`, while a 100 by 100 1-origin matrix might have the bounds `((1,1),(100,100))`. (In many other languages, such bounds would be written in a

form like `1:100, 1:100`, but the present form fits the type system better, since each bound is of the same type as a general index.)

The `range` operation takes a bounds pair and produces the list of indices lying between those bounds, in index order. For example,

```
range (0,4)    ⇒    [0,1,2,3,4]

range ((0,0),(1,2))    ⇒    [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)]
```

The `inRange` predicate determines whether an index lies between a given pair of bounds. (For a tuple type, this test is performed component-wise.) Finally, the `index` operation allows a particular element of an array to be addressed: given a bounds pair and an in-range index, the operation yields the zero-origin ordinal of the index within the range; for example:

```
index (1,9) 2    ⇒    1

index ((0,0),(1,2)) (1,1)    ⇒    4
```

13.2 Array Creation

Haskell's monolithic array creation function forms an array from a pair of bounds and a list of index-value pairs (an *association list*):

```
array                   :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

Here, for example, is a definition of an array of the squares of numbers from 1 to 100:

```
squares                 =  array (1,100) [(i, i*i) | i <- [1..100]]
```

This array expression is typical in using a list comprehension for the association list; in fact, this usage results in array expressions much like the *array comprehensions* of the language Id [6].

Array subscripting is performed with the infix operator `!`, and the bounds of an array can be extracted with the function `bounds`:

```
squares!7    ⇒    49

bounds squares    ⇒    (1,100)
```

We might generalize this example by parameterizing the bounds and the function to be applied to each index:

```
mkArray                 :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bnds          =  array bnds [(i, f i) | i <- range bnds]
```

Thus, we could define `squares` as `mkArray (\i -> i * i) (1,100)`.

Many arrays are defined recursively; that is, with the values of some elements depending on the values of others. Here, for example, we have a function returning an array of Fibonacci numbers:

```
fibs    :: Int -> Array Int Int
fibs n  =  a  where a = array (0,n) ([(0, 1), (1, 1)] ++
                                     [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

Another example of such a recurrence is the n by n *wavefront* matrix, in which elements of the first row and first column all have the value *1* and other elements are sums of their neighbors to the west, northwest, and north:

```
wavefront       :: Int -> Array (Int,Int) Int
wavefront n     =  a  where
                   a = array ((1,1),(n,n))
                        ([((1,j), 1) | j <- [1..n]] ++
                         [((i,1), 1) | i <- [2..n]] ++
                         [((i,j), a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j))
                                     | i <- [2..n], j <- [2..n]])
```

The wavefront matrix is so called because in a parallel implementation, the recurrence dictates that the computation can begin with the first row and column in parallel and proceed as a wedge-shaped wave, traveling from northwest to southeast. It is important to note, however, that no order of computation is specified by the association list.

In each of our examples so far, we have given a unique association for each index of the array and only for the indices within the bounds of the array, and indeed, we must do this in general for an array be fully defined. An association with an out-of-bounds index results in an error; if an index is missing or appears more than once, however, there is no immediate error, but the value of the array at that index is then undefined, so that subscripting the array with such an index yields an error.

13.3 Accumulation

We can relax the restriction that an index appear at most once in the association list by specifying how to combine multiple values associated with a single index; the result is called an *accumulated array*:

```
accumArray :: (Ix a) -> (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b
```

The first argument of `accumArray` is the *accumulating function*, the second is an initial value (the same for each element of the array), and the remaining arguments are bounds and an association list, as with the `array` function. Typically, the accumulating function is `(+)`, and the initial value, zero; for example, this function takes a pair of bounds and a list of values (of an index type) and yields a histogram; that is, a table of the number of occurrences of each value within the bounds:

```
hist            :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is    =  accumArray (+) 0 bnds [(i, 1) | i <- is, inRange bnds i]
```

Suppose we have a collection of measurements on the interval $[a, b)$, and we want to divide the interval into decades and count the number of measurements within each:

```
decades         :: (RealFrac a) => a -> a -> [a] -> Array Int Int
decades a b     =  hist (0,9) . map decade
                   where decade x = floor ((x - a) * s)
                         s        = 10 / (b - a)
```

13.4 Incremental updates

In addition to the monolithic array creation functions, Haskell also has an incremental array update function, written as the infix operator `//`; the simplest case, an array `a` with element `i` updated to `v`, is written `a // [(i, v)]`. The reason for the square brackets is that the left argument of `(//)` is an association list, usually containing a proper subset of the indices of the array:

```
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

As with the `array` function, the indices in the association list must be unique for the values to be defined. For example, here is a function to interchange two rows of a matrix:

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c -> Array (a,b) c
swapRows i i' a =  a // ([((i ,j), a!(i',j)) | j <- [jLo..jHi]] ++
                        [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

The concatenation here of two separate list comprehensions over the same list of `j` indices is, however, a slight inefficiency; it's like writing two loops where one will do in an imperative language. Never fear, we can perform the equivalent of a loop fusion optimization in Haskell:

```
swapRows i i' a =  a // [assoc | j <- [jLo..jHi],
                                 assoc <- [((i ,j), a!(i',j)),
                                           ((i',j), a!(i, j))] ]
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

13.5 An example: Matrix Multiplication

We complete our introduction to Haskell arrays with the familiar example of matrix multiplication, taking advantage of overloading to define a fairly general function. Since only multiplication and addition on the element type of the matrices is involved, we get a function that multiplies matrices of any numeric type unless we try hard not to. Additionally, if we are careful to apply only `(!)` and the operations of `Ix` to indices, we get genericity over index types, and in fact, the four row and column index types need not all be the same. For simplicity, however, we require that the left column indices and right row indices be of the same type, and moreover, that the bounds be equal:

```
matMult         :: (Ix a, Ix b, Ix c, Num d) =>
                   Array (a,b) d -> Array (b,c) d -> Array (a,c) d
matMult x y     =  array resultBounds
                          [((i,j), sum [x!(i,k) * y!(k,j) | k <- range (lj,uj)])
                                       | i <- range (li,ui),
                                         j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

As an aside, we can also define `matMult` using `accumArray`, resulting in a presentation that more closely resembles the usual formulation in an imperative language:

```
matMult x y     =  accumArray (+) 0 resultBounds
                              [((i,j), x!(i,k) * y!(k,j))
                                      | i <- range (li,ui),
                                        j <- range (lj',uj')
                                        k <- range (lj,uj)  ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

We can generalize further by making the function higher-order, simply replacing `sum` and `(*)` by functional parameters:

```
genMatMult      :: (Ix a, Ix b, Ix c) =>
                   ([f] -> g) -> (d -> e -> f) ->
                   Array (a,b) d -> Array (b,c) e -> Array (a,c) g
genMatMult sum' star x y  =
      array resultBounds
            [((i,j), sum' [x!(i,k) `star` y!(k,j) | k <- range (lj,uj)])
                                | i <- range (li,ui),
                                  j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

APL fans will recognize the usefulness of functions like the following:

```
genMatMult maximum (-)
genMatMult and (==)
```

With the first of these, the arguments are numeric matrices, and the (i,j)-th element of the result is the maximum difference between corresponding elements of the i-th row and j-th column of the inputs. In the second case, the arguments are matrices of any equality type, and the result is a Boolean matrix in which element (i,j) is `True` if and only if the i-th row of the first argument and j-th column of the second are equal as vectors.

Notice that the element types of `genMatMult` need not be the same, but merely appropriate for the function parameter `star`. We could generalize still further by dropping the requirement that the first column index and second row index types be the same; clearly, two matrices could be considered conformable as long as the lengths of the columns of the first and the rows of the second are equal. The reader may wish to derive this still more general version. (**Hint:** Use the `index` operation to determine the lengths.)

14 The Next Stage

A large collection of Haskell resources is available on the web at `haskell.org`. Here you will find compilers, demos, papers, and much valuable information about Haskell and functional programming. Haskell compilers or interpreters run on almost all hardware and operating systems. The Hugs system is both small and portable – it is an excellent vehicle for learning Haskell.

15 Acknowledgements

Thanks to Patricia Fasel and Mark Mundt at Los Alamos, and Nick Carriero, Charles Consel, Amir Kishon, Sandra Loosemore, Martin Odersky, and David Rochberg at Yale University for their quick readings of earlier drafts of this manuscript. Special thanks to Erik Meijer for his extensive comments on the new material added for version 1.4 of this tutorial.

References

[1] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, New York, 1998.

[2] A.Davie. *Introduction to Functional Programming System Using Haskell.* Cambridge University Press, 1992.

[3] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[4] Simon Peyton Jones (editor). Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language. *Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106*, Feb 1999.

[5] Simon Peyton Jones (editor) The Haskell 98 Library Report. *Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1105*, Feb 1999.

[6] R.S. Nikhil. Id (version 90.0) reference manual. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[7] J. Rees and W. Clinger (eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[8] G.L. Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington, Mass., 1984.

[9] P. Wadler. How to replace failure by a list of successes. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture, LNCS Vol. 201*, pages 113–128. Springer Verlag, 1985.

[10] P. Wadler. Monads for Functional Programming In *Advanced Functional Programming* , Springer Verlag, LNCS 925, 1995.

Foundations of Functional Programming

Computer Science Tripos Part IB
Easter Term

Lawrence C Paulson
Computer Laboratory
University of Cambridge

lcp@cl.cam.ac.uk

Contents

1 Introduction

This course is concerned with the λ-calculus and its close relative, combinatory logic. The λ-calculus is important to functional programming and to computer science generally:

1. *Variable binding* and *scoping* in block-structured languages can be modelled.

2. Several function calling mechanisms — *call-by-name*, *call-by-value*, and *call-by-need* — can be modelled. The latter two are also known as *strict evaluation* and *lazy evaluation*.

3. The λ-calculus is Turing universal, and is probably the most natural model of computation. *Church's Thesis* asserts that the 'computable' functions are precisely those that can be represented in the λ-calculus.

4. All the usual *data structures* of functional programming, including infinite lists, can be represented. Computation on infinite objects can be defined formally in the λ-calculus.

5. Its notions of *confluence* (Church-Rosser property), *termination*, and *normal form* apply generally in rewriting theory.

6. *Lisp*, one of the first major programming languages, was inspired by the λ-calculus. Many functional languages, such as ML, consist of little more than the λ-calculus with additional syntax.

7. The two main implementation methods, the *SECD machine* (for strict evaluation) and *combinator reduction* (for lazy evaluation) exploit properties of the λ-calculus.

8. The λ-calculus and its extensions can be used to develop better type systems, such as *polymorphism*, and to investigate theoretical issues such as *program synthesis*.

9. *Denotational semantics*, which is an important method for formally specifying programming languages, employs the λ-calculus for its notation.

Hindley and Seldin [6] is a concise introduction to the λ-calculus and combinators. Gordon [5] is oriented towards computer science, overlapping closely with this course. Barendregt [1] is the last word on the λ-calculus.

Acknowledgements. Reuben Thomas pointed out numerous errors in a previous version of these notes.

1.1 The λ-Calculus

Around 1924, Schönfinkel developed a simple theory of functions. In 1934, Church introduced the λ-calculus and used it to develop a formal set theory, which turned out to be inconsistent. More successfully, he used it to formalize the syntax of Whitehead and Russell's massive *Principia Mathematica*. In the 1940s, Haskell B. Curry introduced *combinatory logic*, a variable-free theory of functions.

More recently, Roger Hindley developed what is now known as *type inference*. Robin Milner extended this to develop the polymorphic type system of ML, and published a proof that a well-typed program cannot suffer a run-time type error. Dana Scott developed models of the λ-calculus. With his *domain theory*, he and Christopher Strachey introduced denotational semantics.

Peter Landin used the λ-calculus to analyze Algol 60, and introduced ISWIM as a framework for future languages. His SECD machine, with extensions, was used to implement ML and other strict functional languages. Christopher Wadsworth developed *graph reduction* as a method for performing lazy evaluation of λ-expressions. David Turner applied graph reduction to *combinators*, which led to efficient implementations of lazy evaluation.

Definition 1 The *terms* of the λ-calculus, known as λ*-terms*, are constructed recursively from a given set of *variables* x, y, z, ... They may take one of the following forms:

x	variable
$(\lambda x.M)$	abstraction, where M is a term
(MN)	application, where M and N are terms

We use capital letters like L, M, N, ... for terms. We write $M \equiv N$ to state that M and N are identical λ-terms. The equality between λ-terms, $M = N$, will be discussed later.

1.2 Variable Binding and Substitution

In $(\lambda x.M)$, we call x the *bound variable* and M the *body*. Every occurrence of x in M is *bound* by the abstraction. An occurrence of a variable is *free* if it is not bound by some enclosing abstraction. For example, x occurs bound and y occurs free in $(\lambda z.(\lambda x.(yx))$.

Notations involving free and bound variables exist throughout mathematics. Consider the integral $\int_a^b f(x)dx$, where x is bound, and the product $\Pi_{k=0}^{n} p(k)$, where k is bound. The quantifiers $\forall$ and $\exists$ also bind variables.

The abstraction $(\lambda x.M)$ is intended to represent the function f such that $f(x) = M$ for all x. Applying f to N yields the result of substituting N for all free occurrences of x in M. Two examples are

$(\lambda x.x)$ The *identity* function, which returns its argument unchanged. It is usually called $\mathbf{I}$.

$(\lambda y.x)$ A *constant* function, which returns x when applied to any argument.

Let us make these concepts precise.

Definition 2 $\mathrm{BV}(M)$, the set of all *bound variables* in M, is given by

$$\begin{aligned} \mathrm{BV}(x) &= \emptyset \\ \mathrm{BV}(\lambda x.M) &= \mathrm{BV}(M) \bigcup \{x\} \\ \mathrm{BV}(MN) &= \mathrm{BV}(M) \bigcup \mathrm{BV}(N) \end{aligned}$$

Definition 3 $\mathrm{FV}(M)$, the set of all *free variables* in M, is given by

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) - \{x\} \\ \mathrm{FV}(MN) &= \mathrm{FV}(M) \bigcup \mathrm{FV}(N) \end{aligned}$$

Definition 4 $M[L/y]$, the result of substituting L for all free occurrences of y in M, is given by

$$\begin{aligned} x[L/y] &\equiv \begin{cases} L & \text{if } x \equiv y \\ x & \text{otherwise} \end{cases} \\ (\lambda x.M)[L/y] &\equiv \begin{cases} (\lambda x.M) & \text{if } x \equiv y \\ (\lambda x.M[L/y]) & \text{otherwise} \end{cases} \\ (MN)[L/y] &\equiv (M[L/y]\ N[L/y]) \end{aligned}$$

The notations defined above are not themselves part of the λ-calculus. They belong to the metalanguage: they are for talking about the λ-calculus.

1.3 Avoiding Variable Capture in Substitution

Substitution must not disturb variable binding. Consider the term $(\lambda x.(\lambda y.x))$. It should represent the function that, when applied to an argument N, returns the constant function $(\lambda y.N)$. Unfortunately, this does not work if $N \equiv y$; we have defined substitution such that $(\lambda y.x)[y/x] \equiv (\lambda y.y)$. Replacing x by y in the constant function transforms it into the identity function. The free occurrence of x turns into a bound occurrence of y — an example of *variable capture*. If this were allowed to happen, the λ-calculus would be inconsistent. The substitution $M[N/x]$ is safe provided the bound variables of M are disjoint from the free variables of N:

$$\mathrm{BV}(M) \bigcap \mathrm{FV}(N) = \emptyset.$$

We can always rename the bound variables of M, if necessary, to make this condition true. In the example above, we could change $(\lambda y.x)$ into $(\lambda z.x)$, then obtain the correct substitution $(\lambda z.x)[y/x] \equiv (\lambda z.y)$; the result is indeed a constant function.

1.4 Conversions

The idea that λ-abstractions represent functions is formally expressed through conversion rules for manipulating them. There are α-conversions, β-conversions and η-conversions.

The α-conversion $(\lambda x.M) \to_\alpha (\lambda y.M[y/x])$ renames the abstraction's bound variable from x to y. It is valid provided y does not occur (free or bound) in M. For example, $(\lambda x.(xz)) \to_\alpha (\lambda y.(yz))$. We shall usually ignore the distinction between terms that could be made identical by performing α-conversions.

The β-conversion $((\lambda x.M)N) \to_\beta M[N/x]$ substitutes the argument, N, into the abstraction's body, M. It is valid provided $\mathrm{BV}(M) \bigcap \mathrm{FV}(N) = \emptyset$. For example, $(\lambda x.(xx))(yz) \to_\beta ((yz)(yz))$. Here is another example: $((\lambda z.(zy))(\lambda x.x)) \to_\beta ((\lambda x.x)y) \to_\beta y$.

The η-conversion $(\lambda x.(Mx)) \to_\eta M$ collapses the trivial function $(\lambda x.(Mx))$ down to M. It is valid provided $x \notin \mathrm{FV}(M)$. Thus, M does not depend on x; the abstraction does nothing but apply M to its argument. For example, $(\lambda x.((zy)x)) \to_\eta (zy)$.

Observe that the functions $(\lambda x.(Mx))$ and M always return the same answer, (MN), when applied to any argument N. The η-conversion rule embodies a principle of *extensionality*: two functions are equal if they always return equal results given equal arguments. In some situations, this principle (and η-conversions) are dispensed with.

1.5 Reductions

We say that $M \to N$, or M *reduces to* N, if $M \to_\beta N$ or $M \to_\eta N$. (Because α-conversions are not directional, and are not interesting, we generally ignore them.) The reduction $M \to N$ may consist of applying a conversion to some subterm of M in order to create N. More formally, we could introduce inference rules for $\to$:

$$\frac{M \to M'}{(\lambda x.M) \to (\lambda x.M')} \qquad \frac{M \to M'}{(MN) \to (M'N)} \qquad \frac{M \to M'}{(LM) \to (LM')}$$

If a term admits no reductions then it is in *normal form*. For example, $\lambda xy.y$ and xyz are in normal form. To *normalize* a term means to apply reductions until a normal form is reached. A term *has a normal form* if it can be reduced to a term in normal form. For example, $(\lambda x.x)y$ is not in normal form, but it has the normal form y.

Many λ-terms cannot be reduced to normal form. For instance, $(\lambda x.xx)(\lambda x.xx)$ reduces to itself by β-conversion. Although it is unaffected by the reduction, it is certainly not in normal form. This term is usually called Ω.

1.6 Curried Functions

The λ-calculus has only functions of one argument. A function with multiple arguments is expressed using a function whose result is another function.

For example, suppose that L is a term containing only x and y as free variables, and we wish to formalize the function $f(x, y) = L$. The abstraction $(\lambda y.L)$ contains x free; for each x, it stands for a function over y. The abstraction $(\lambda x.(\lambda y.L))$ contains no free variables; when applied to the arguments M and N, the result is obtained by replacing x by M and y by N in L. Symbolically, we perform two β-reductions (any necessary α-conversions are omitted):

$$(((\lambda x.(\lambda y.L))M)N) \to_\beta ((\lambda y.L[M/x])N) \to_\beta L[M/x][N/y]$$

This technique is known as *currying* after Haskell B. Curry, and a function expressed using nested λs is known as a *curried function*. In fact, it was introduced by Schönfinkel. Clearly, it works for any number of arguments.

Curried functions are popular in functional programming because they can be applied to their first few arguments, returning functions that are useful in themselves.

1.7 Bracketing Conventions

Abbreviating nested abstractions and applications will make curried functions easier to write. We shall abbreviate

$$\begin{array}{rcl}(\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.M)\ldots)) & \text{as} & (\lambda x_1 x_2 \ldots x_n.M) \\ (\ldots(M_1 M_2)\ldots M_n) & \text{as} & (M_1 M_2 \ldots M_n)\end{array}$$

Finally, we drop outermost parentheses and those enclosing the body of an abstraction. For example,

$(\lambda x.(x(\lambda y.(yx))))$ can be written as $\lambda x.x(\lambda y.yx)$.

It is vital understand how bracketing works. We have the reduction

$$\lambda z.(\lambda x.M)N \to_\beta \lambda z.M[N/x]$$

but the similar term $\lambda z.z(\lambda x.M)N$ admits no reductions except those occurring within M and N, because $\lambda x.M$ is not being applied to anything. Here is what the application of a curried function (see above) looks like with most brackets omitted:

$$(\lambda xy.L)MN \to_\beta (\lambda y.L[M/x])N \to_\beta L[M/x][N/y]$$

Note that $\lambda x.MN$ abbreviates $\lambda x.(MN)$ rather than $(\lambda x.M)N$. Also, xyz abbreviates $(xy)z$ rather than $x(yz)$.

Exercise 1 What happens in the reduction of $(\lambda xy.L)MN$ if y is free in M?

Exercise 2 Give two different reduction sequences that start at $(\lambda x.(\lambda y.xy)z)y$ and end with a normal form. (These normal forms must be identical: see below.)

2 Equality and Normalization

The λ-calculus is an equational theory: it consists of rules for proving that two λ-terms are equal. A key property is that two terms are equal just if they both can be reduced to the same term.

2.1 Multi-Step Reduction

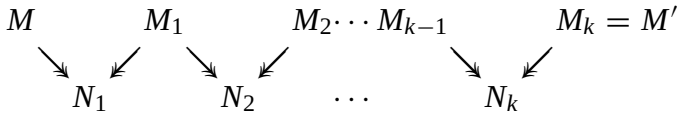
Strictly speaking, $M \to N$ means that M reduces to N by exactly one reduction step, possibly applied to a subterm of M. Frequently, we are interested in whether M can be reduced to N by any number of steps. Write $M \twoheadrightarrow N$ if

$$M \to M_1 \to M_2 \to \cdots \to M_k \equiv N \qquad (k \geq 0)$$

For example, $((\lambda z.(zy))(\lambda x.x)) \twoheadrightarrow y$. Note that $\twoheadrightarrow$ is the relation $\to^*$, the reflexive/transitive closure of $\to$.

2.2 Equality Between λ-Terms

Informally, $M = M'$ if M can be transformed into M' by performing zero or more reductions and expansions. (An *expansion* is the inverse of a reduction, for instance $y \leftarrow (\lambda x.x)y$.) A typical picture is the following:



For example, $a((\lambda y.by)c) = (\lambda x.ax)(bc)$ because both sides reduce to $a(bc)$. Note that $=$ is the relation $(\to \cup \to^{-1})^*$, the least equivalence relation containing $\to$.

Intuitively, $M = M'$ means that M and M' have the same value. Equality, as defined here, satisfies all the standard properties. First of all, it is an *equivalence relation* — it satisfies the reflexive, symmetric and associative laws:

$$M = M \qquad \frac{M = N}{N = M} \qquad \frac{L = M \quad M = N}{L = N}$$

Furthermore, it satisfies congruence laws for each of the ways of constructing λ-terms:

$$\frac{M = M'}{(\lambda x.M) = (\lambda x.M')} \qquad \frac{M = M'}{(MN) = (M'N)} \qquad \frac{M = M'}{(LM) = (LM')}$$

The six properties shown above are easily checked by constructing the appropriate diagrams for each equality. They imply that two terms will be equal if we construct them in the same way starting from equal terms. Put another way, if $M = M'$ then replacing M by M' in a term yields an equal term.

Definition 5 *Equality of λ-terms* is the least relation satisfying the six rules given above.

2.3 The Church-Rosser Theorem

This fundamental theorem states that reduction in the λ-calculus is *confluent*: no two sequences of reductions, starting from one λ-term, can reach distinct normal forms. The normal form of a term is independent of the order in which reductions are performed.

Theorem 6 (Church-Rosser) *If $M = N$ then there exists L such that $M \twoheadrightarrow L$ and $N \twoheadrightarrow L$.*
Proof See Barendregt [1] or Hindley and Seldin [6].

For instance, $(\lambda x.ax)((\lambda y.by)c)$ has two different reduction sequences, both leading to the same normal form. The affected subterm is underlined at each step:

$$\underline{(\lambda x.ax)((\lambda y.by)c)} \to a(\underline{(\lambda y.by)c}) \to a(bc)$$
$$(\lambda x.ax)(\underline{(\lambda y.by)c}) \to \underline{(\lambda x.ax)(bc)} \to a(bc)$$

The theorem has several important consequences.

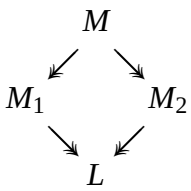
- If $M = N$ and N is in normal form, then $M \twoheadrightarrow N$; if a term can transform into normal form using reductions and expansions, then the normal form can be reached by reductions alone.

- If $M = N$ where both terms are in normal form, then $M \equiv N$ (up to renaming of bound variables). Conversely, if M and N are in normal form and are distinct, then $M \neq N$; there is no way of transforming M into N. For example, $\lambda xy.x \neq \lambda xy.y$.

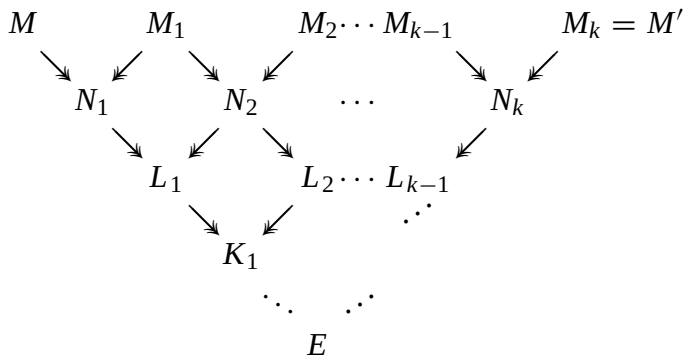
An equational theory is *inconsistent* if all equations are provable. Thanks to the Church-Rosser Theorem, we know that the λ-calculus is consistent. There is no way we could reach two different normal forms by following different reduction strategies. Without this property, the λ-calculus would be of little relevance to computation.

2.4 The Diamond Property

The key step in proving the Church-Rosser Theorem is demonstrating the diamond property — if $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$ then there exists a term L such that $M_1 \twoheadrightarrow L$ and $M_2 \twoheadrightarrow L$. Here is the diagram:

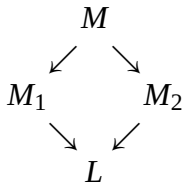


The diamond property is vital: it says that no matter how far we go reducing a term by two different strategies it will always be possible to come together again by further reductions. As for the Church-Rosser Theorem, look again at the diagram for $M = M'$ and note that we can tile the region underneath with diamonds, eventually reaching a common term:

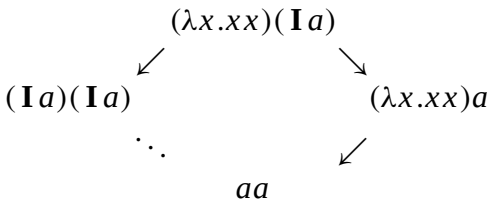


2.5 Proving the Diamond Property

Note that $\rightarrow$ (one-step reduction) does *not* satisfy the diamond property



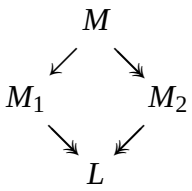
Consider the term $(\lambda x.xx)(\mathbf{I}\,a)$, where $\mathbf{I} \equiv \lambda x.x$. In one step, it reduces to $(\lambda x.xx)a$ or to $(\mathbf{I}\,a)(\mathbf{I}\,a)$. These both reduce eventually to aa, but there is no way to complete the diamond with a single-step reduction:



The problem, of course, is that $(\lambda x.xx)$ replicates its argument, which must then be reduced twice. Note also that the difficult cases involve one possible

reduction contained inside another. Reductions that do not overlap, such as $M \to M'$ and $N \to N'$ in the term $x\,M\,N$, commute trivially to produce $x\,M'\,N'$.

The diamond property for $\twoheadrightarrow$ can be proved with the help of a 'strip lemma', which considers the case where $M \to M_1$ (in *one* step) and also $M \twoheadrightarrow M_2$ (possibly *many* steps):



The 'strips' can then be pasted together to complete a diamond. The details involve an extremely tedious case analysis of the possible reductions from various forms of terms.

2.6 Possibility of Nontermination

Although different reduction sequences cannot yield different normal forms, they can yield completely different outcomes: one could terminate while the other runs forever! Typically, if M has a normal form and admits an infinite reduction sequence, it contains a subterm L having no normal form, and L can be erased by a reduction.

For example, recall that Ω reduces to itself, where $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. The reduction

$$\underline{(\lambda y.a)\Omega} \to a$$

reaches normal form, erasing the Ω. This corresponds to a *call-by-name* treatment of functions: the argument is not reduced but substituted 'as is' into the body of the abstraction.

Attempting to normalize the argument generates a nonterminating reduction sequence:

$$(\lambda y.a)\Omega \to (\lambda y.a)\Omega \to \cdots$$

Evaluating the argument before substituting it into the body corresponds to a *call-by-value* treatment of function application. In this example, the call-by-value strategy never reaches the normal form.

2.7 Normal Order Reduction

The *normal order* reduction strategy is, at each step, to perform the leftmost outermost β-reduction. (The η-reductions can be left until last.) *Leftmost* means, for

instance, to reduce L before N in $L\,N$. *Outermost* means, for instance, to reduce $(\lambda x.M)N$ before reducing M or N.

Normal order reduction corresponds to call-by-name evaluation. By the Standardization Theorem, it always reaches a normal form if one exists. The proof is omitted. However, note that reducing L first in $L\,N$ may transform L into an abstraction, say $\lambda x.M$. Reducing $(\lambda x.M)N$ may erase N.

2.8 Lazy Evaluation

From a theoretical standpoint, normal order reduction is the optimal, since it always yields a normal form if one exists. For practical computation, it is hopelessly inefficient. Assume that we have a coding of the natural numbers (for which see the next section!) and define a squaring function $\mathbf{sqr} \equiv \lambda n.\,\mathbf{mult}\,nn$. Then

$$\mathbf{sqr}\,(\mathbf{sqr}\,N) \to \mathbf{mult}\,(\mathbf{sqr}\,N)(\mathbf{sqr}\,N) \to \mathbf{mult}\,(\mathbf{mult}\,N\,N)(\mathbf{mult}\,N\,N)$$

and we will have to evaluate four copies of the term N! Call-by-value would have evaluated N (only once) beforehand, but, as we have seen, it can result in nontermination.

Note: multi-letter identifiers (like **sqr**) are set in bold type, or underlined, in order to prevent confusion with a series of separate variables (like sqr).

Lazy evaluation, or *call-by-need*, never evaluates an argument more than once. An argument is not evaluated unless the value is actually required to produce the answer; even then, the argument is only evaluated to the extent needed (thereby allowing infinite lists). Lazy evaluation can be implemented by representing the term by a graph rather than a tree. Each shared graph node represents a subterm whose value is needed more than once. Whenever that subterm is reduced, the result overwrites the node, and the other references to it will immediately have access to the replacement.

Graph reduction is inefficient for the λ-calculus because subterms typically contain free variables. During each β-reduction, the abstraction's body must be copied. Graph reduction works much better for combinators, where there are no variables. We shall return to this point later.

3 Encoding Data in the λ-Calculus

The λ-calculus is expressive enough to encode boolean values, ordered pairs, natural numbers and lists — all the data structures we may desire in a functional program. These encodings allow us to model virtually the whole of functional programming within the simple confines of the λ-calculus.

The encodings may not seem to be at all natural, and they certainly are not computationally efficient. In this, they resemble Turing machine encodings and programs. Unlike Turing machine programs, the encodings are themselves of mathematical interest, and return again and again in theoretical studies. Many of them involve the idea that the data can carry its control structure with it.

3.1 The Booleans

An encoding of the booleans must define the terms **true**, **false** and **if**, satisfying (for all M and N)

$$\begin{aligned} \textbf{if true}\, MN &= M \\ \textbf{if false}\, MN &= N. \end{aligned}$$

The following encoding is usually adopted:

$$\begin{aligned} \textbf{true} &\equiv \lambda xy.x \\ \textbf{false} &\equiv \lambda xy.y \\ \textbf{if} &\equiv \lambda pxy.pxy \end{aligned}$$

We have $\textbf{true} \neq \textbf{false}$ by the Church-Rosser Theorem, since **true** and **false** are distinct normal forms. As it happens, **if** is not even necessary. The truth values are their own conditional operators:

$$\begin{aligned} \textbf{true}\, MN &\equiv (\lambda xy.x)MN \twoheadrightarrow M \\ \textbf{false}\, MN &\equiv (\lambda xy.y)MN \twoheadrightarrow N \end{aligned}$$

These reductions hold for all terms M and N, whether or not they possess normal forms. Note that $\textbf{if}\, LMN \twoheadrightarrow LMN$; it is essentially an identity function on L. The equations given above even hold as reductions:

$$\begin{aligned} \textbf{if true}\, MN &\twoheadrightarrow M \\ \textbf{if false}\, MN &\twoheadrightarrow N. \end{aligned}$$

All the usual operations on truth values can be defined as conditional operator. Here are negation, conjunction and disjunction:

$$\begin{aligned} \textbf{and} &\equiv \lambda pq.\, \textbf{if}\, p\, q\, \textbf{false} \\ \textbf{or} &\equiv \lambda pq.\, \textbf{if}\, p\, \textbf{true}\, q \\ \textbf{not} &\equiv \lambda p.\, \textbf{if}\, p\, \textbf{false}\, \textbf{true} \end{aligned}$$

3.2 Ordered Pairs

Assume that **true** and **false** are defined as above. The function **pair**, which constructs pairs, and the projections **fst** and **snd**, which select the components of a pair, are encoded as follows:

$$\begin{aligned} \mathbf{pair} &\equiv \lambda xyf.\, f\,x\,y \\ \mathbf{fst} &\equiv \lambda p.\, p\,\mathbf{true} \\ \mathbf{snd} &\equiv \lambda p.\, p\,\mathbf{false} \end{aligned}$$

Clearly, $\mathbf{pair}\,M\,N \twoheadrightarrow \lambda f.\, f\,M\,N$, packaging M and N together. A pair may be applied to any 2-place function of the form $\lambda xy.L$, returning $L[M/x][N/y]$; thus, each pair is its own unpackaging operation. The projections work by this unpackaging operation (which, perhaps, is more convenient in programming than are the projections themselves!):

$$\begin{aligned} \mathbf{fst}\,(\,\mathbf{pair}\,M\,N) &\twoheadrightarrow \mathbf{fst}\,(\lambda f.\, f\,M\,N) \\ &\to (\lambda f.\, f\,M\,N)\,\mathbf{true} \\ &\to \mathbf{true}\,M\,N \\ &\twoheadrightarrow M \end{aligned}$$

Similarly, $\mathbf{snd}\,(\,\mathbf{pair}\,M\,N) \twoheadrightarrow N$. Observe that the components of $\mathbf{pair}\,M\,N$ are completely independent; either may be extracted even if the other has no normal form.

Ordered n-tuples could be defined analogously, but nested pairs are a simpler encoding.

3.3 The Natural Numbers

The following encoding of the natural numbers is the original one developed by Church. Alternative encodings are sometimes preferred today, but Church's numerals continue our theme of putting the control structure in with the data structure. Such encodings are elegant; moreover, they work in the second-order λ-calculus (presented in the Types course by Andrew Pitts).

Define

$$\begin{aligned} \underline{0} &\equiv \lambda f x.x \\ \underline{1} &\equiv \lambda f x.\, f\,x \\ \underline{2} &\equiv \lambda f x.\, f\,(f\,x) \\ &\vdots \\ \underline{n} &\equiv \lambda f x.\, \underbrace{f\,(\cdots(f}_{n\text{ times}}\,x)\cdots) \end{aligned}$$

Thus, for all $n \geq 0$, the Church numeral $\underline{n}$ is the function that maps f to f^n. Each numeral is an iteration operator.

3.4 Arithmetic on Church Numerals

Using this encoding, addition, multiplication and exponentiation can be defined immediately:

$$\begin{aligned} \mathbf{add} &\equiv \lambda mnfx.mf(nfx) \\ \mathbf{mult} &\equiv \lambda mnfx.m(nf)x \\ \mathbf{expt} &\equiv \lambda mnfx.nmfx \end{aligned}$$

Addition is not hard to check:

$$\begin{aligned} \mathbf{add}\ \underline{m}\ \underline{n} &\twoheadrightarrow \lambda fx.\underline{m}\ f(\underline{n}\ fx) \\ &\twoheadrightarrow \lambda fx.f^m(f^n x) \\ &\equiv \lambda fx.f^{m+n}x \\ &\equiv \underline{m+n} \end{aligned}$$

Multiplication is slightly more difficult:

$$\begin{aligned} \mathbf{mult}\ \underline{m}\ \underline{n} &\twoheadrightarrow \lambda fx.\underline{m}\,(\underline{n}\ f)x \\ &\twoheadrightarrow \lambda fx.(\underline{n}\ f)^m x \\ &\twoheadrightarrow \lambda fx.(f^n)^m x \\ &\equiv \lambda fx.f^{m\times n}x \\ &\equiv \underline{m\times n} \end{aligned}$$

These derivations hold for all Church numerals $\underline{m}$ and $\underline{n}$, but not for all terms M and N.

Exercise 3 Show that **expt** performs exponentiation on Church numerals.

3.5 The Basic Operations for Church Numerals

The operations defined so far are not sufficient to define all computable functions on the natural numbers; what about subtraction? Let us begin with some simpler definitions: the successor function and the zero test.

$$\begin{aligned} \mathbf{suc} &\equiv \lambda nfx.f(nfx) \\ \mathbf{iszero} &\equiv \lambda n.n(\lambda x.\,\mathbf{false}\,)\,\mathbf{true} \end{aligned}$$

The following reductions hold for every Church numeral $\underline{n}$:

$$\begin{array}{rcl} \textbf{suc}\ \underline{n} & \twoheadrightarrow & \underline{n+1} \\ \textbf{iszero}\ \underline{0} & \twoheadrightarrow & \textbf{true} \\ \textbf{iszero}\ (\underline{n+1}) & \twoheadrightarrow & \textbf{false} \end{array}$$

For example,

$$\begin{array}{rcl} \textbf{iszero}\ (\underline{n+1}) & \twoheadrightarrow & \underline{n+1}\ (\lambda x.\,\textbf{false})\ \textbf{true} \\ & \twoheadrightarrow & (\lambda x.\,\textbf{false})^{n+1}\ \textbf{true} \\ & \equiv & (\lambda x.\,\textbf{false})((\lambda x.\,\textbf{false})^{n}\ \textbf{true}) \\ & \to & \textbf{false} \end{array}$$

The predecessor function and subtraction are encoded as follows:

$$\begin{array}{rcl} \textbf{prefn} & \equiv & \lambda f p.\,\textbf{pair}\,(f\,(\textbf{fst}\ p))\,(\textbf{fst}\ p) \\ \textbf{pre} & \equiv & \lambda n f x.\,\textbf{snd}\,(n(\textbf{prefn}\ f)(\textbf{pair}\,x x)) \\ \textbf{sub} & \equiv & \lambda m n.n\textbf{pre}\,m \end{array}$$

Defining the predecessor function is difficult when each numeral is an iterator. We must reduce an $n+1$ iterator to an n iterator. Given f and x, we must find some g and y such that $g^{n+1}y$ computes $f^n x$. A suitable g is a function on pairs that maps (x, z) to $(f(x), x)$; then

$$g^{n+1}(x, x) = (f^{n+1}(x),\ f^n(x)).$$

The pair behaves like a one-element delay line.

Above, **prefn** f constructs the function g. Verifying the following reductions should be routine:

$$\begin{array}{rcl} \textbf{pre}\,(\underline{n+1}) & \twoheadrightarrow & \underline{n} \\ \textbf{pre}\,(\underline{0}) & \twoheadrightarrow & \underline{0} \end{array}$$

For subtraction, **sub** $\underline{m}$ $\underline{n}$ computes the nth predecessor of $\underline{m}$.

Exercise 4 Show that $\lambda m n.m\ \textbf{suc}\ n$ performs addition on Church numerals.

3.6 Lists

Church numerals could be generalized to represent lists. The list $[x_1, x_2, \ldots, x_n]$ would essentially be represented by the function that takes f and y to

$f\,x_1\,(f\,x_2\,\ldots\,(f\,x_n\,y)\ldots)$. Such lists would carry their own control structure with them.

As an alternative, let us represent lists rather as Lisp and ML do — via pairing. This encoding is easier to understand because it is closer to real implementations. The list $[x_1, x_2, \ldots, x_n]$ will be represented by $x_1 :: x_2 :: \ldots :: \mathbf{nil}$. To keep the operations as simple as possible, we shall employ two levels of pairing. Each 'cons cell' $x :: y$ will be represented by $(\mathbf{false}, (x, y))$, where the **false** is a distinguishing tag field. By rights, **nil** should be represented by a pair whose first component is **true**, such as $(\mathbf{true}, \mathbf{true})$, but a simpler definition happens to work. In fact, we could dispense with the tag field altogether.

Here is our encoding of lists:

$$\begin{aligned}
\mathbf{nil} &\equiv \lambda z.z \\
\mathbf{cons} &\equiv \lambda xy.\,\mathbf{pair}\;\mathbf{false}\,(\mathbf{pair}\,x\,y) \\
\mathbf{null} &\equiv \mathbf{fst} \\
\mathbf{hd} &\equiv \lambda z.\,\mathbf{fst}\,(\mathbf{snd}\,z) \\
\mathbf{tl} &\equiv \lambda z.\,\mathbf{snd}\,(\mathbf{snd}\,z)
\end{aligned}$$

The following properties are easy to verify; they hold for all terms M and N:

$$\begin{aligned}
\mathbf{null}\;\mathbf{nil} &\twoheadrightarrow \mathbf{true} \\
\mathbf{null}\,(\mathbf{cons}\,M\,N) &\twoheadrightarrow \mathbf{false} \\
\mathbf{hd}\,(\mathbf{cons}\,M\,N) &\twoheadrightarrow M \\
\mathbf{tl}\,(\mathbf{cons}\,M\,N) &\twoheadrightarrow N
\end{aligned}$$

Note that $\mathbf{null}\;\mathbf{nil} \twoheadrightarrow \mathbf{true}$ happens really by chance, while the other laws hold by our operations on pairs.

Recall that laws like $\mathbf{hd}\,(\mathbf{cons}\,M\,N) \twoheadrightarrow M$ and $\mathbf{snd}\,(\mathbf{pair}\,M\,N) \twoheadrightarrow N$ hold for all M and N, even for terms that have no normal forms! Thus, **pair** and **cons** are 'lazy' constructors — they do not 'evaluate their arguments'. Once we introduction recursive definitions, we shall be able to compute with infinite lists.

Exercise 5 Modify the encoding of lists to obtain an encoding of the natural numbers.

4 Writing Recursive Functions in the λ-calculus

Recursion is obviously essential in functional programming. With Church numerals, it is possible to define 'nearly all' computable functions on the natural

numbers.[1] Church numerals have an inbuilt source of repetition. From this, we can derive primitive recursion, which when applied using higher-order functions defines a much larger class than the primitive recursive functions studied in Computation Theory. Ackermann's function is not primitive recursive in the usual sense, but we can encode it using Church numerals. If we put

$$\mathbf{ack} \equiv \lambda m.m(\lambda f n.nf(f\ \underline{1}))\ \mathbf{suc}$$

then we can derive the recursion equations of Ackermann's function, namely

$$\begin{array}{rcl} \mathbf{ack}\ \underline{0}\ \underline{n} & = & \underline{n+1} \\ \mathbf{ack}\ (\underline{m+1})\ \underline{0} & = & \mathbf{ack}\ \underline{m}\ \underline{1} \\ \mathbf{ack}\ (\underline{m+1})(\underline{n+1}) & = & \mathbf{ack}\ \underline{m}\ (\mathbf{ack}\ (\underline{m+1})\ \underline{n}) \end{array}$$

Let us check the first equation:

$$\begin{array}{rcl} \mathbf{ack}\ \underline{0}\ \underline{n} & \twoheadrightarrow & \underline{0}\ (\lambda f n.nf(f\ \underline{1}))\ \mathbf{suc}\ \underline{n} \\ & \twoheadrightarrow & \mathbf{suc}\ \underline{n} \\ & \twoheadrightarrow & \underline{n+1} \end{array}$$

For the other two equations, note that

$$\begin{array}{rcl} \mathbf{ack}\ (\underline{m+1})\ \underline{n} & \twoheadrightarrow & (\underline{m+1})(\lambda f n.nf(f\ \underline{1}))\ \mathbf{suc}\ \underline{n} \\ & \twoheadrightarrow & (\lambda f n.nf(f\ \underline{1}))(\underline{m}\ (\lambda f n.nf(f\ \underline{1}))\ \mathbf{suc})\ \underline{n} \\ & = & (\lambda f n.nf(f\ \underline{1}))(\mathbf{ack}\ \underline{m})\ \underline{n} \\ & \twoheadrightarrow & \underline{n}\ (\mathbf{ack}\ \underline{m})(\mathbf{ack}\ \underline{m}\ \underline{1}) \end{array}$$

We now check

$$\begin{array}{rcl} \mathbf{ack}\ (\underline{m+1})\ \underline{0} & \twoheadrightarrow & \underline{0}\ (\mathbf{ack}\ \underline{m})(\mathbf{ack}\ \underline{m}\ \underline{1}) \\ & \twoheadrightarrow & \mathbf{ack}\ \underline{m}\ \underline{1} \end{array}$$

and

$$\begin{array}{rcl} \mathbf{ack}\ (\underline{m+1})(\underline{n+1}) & \twoheadrightarrow & \underline{n+1}\ (\mathbf{ack}\ \underline{m})(\mathbf{ack}\ \underline{m}\ \underline{1}) \\ & \twoheadrightarrow & \mathbf{ack}\ \underline{m}\ (\underline{n}\ (\mathbf{ack}\ \underline{m})(\mathbf{ack}\ \underline{m}\ \underline{1})) \\ & = & \mathbf{ack}\ \underline{m}\ (\mathbf{ack}\ (\underline{m+1})\ \underline{n}) \end{array}$$

The key to this computation is the iteration of the function $\mathbf{ack}\ \underline{m}$.

[1] The precise meaning of 'nearly all' involves heavy proof theory, but all 'reasonable' functions are included.

4.1 Recursive Functions using Fixed Points

Our coding of Ackermann's function works, but it hardly could be called perspicuous. Even worse would be the treatment of a function whose recursive calls involved something other than subtracting one from an argument — performing division by repeated subtraction, for example.

General recursion can be derived in the λ-calculus. Thus, we can model all recursive function definitions, even those that fail to terminate for some (or all) arguments. Our encoding of recursion is completely uniform and is independent of the details of the recursive definition and the representation of the data structures (unlike the above version of Ackermann's function, which depends upon Church numerals).

The secret is to use a *fixed point combinator* — a term $\mathbf{Y}$ such that $\mathbf{Y}\,F = F(\mathbf{Y}\,F)$ for all terms F. Let us explain the terminology. A *fixed point* of the function F is any X such that $FX = X$; here, $X \equiv \mathbf{Y}\,F$. A *combinator* is any λ-term containing no free variables (also called a closed term). To code recursion, F represents the body of the recursive definition; the law $\mathbf{Y}\,F = F(\mathbf{Y}\,F)$ permits F to be unfolded as many times as necessary.

4.2 Examples Using Y

We shall encode the factorial function, the append function on lists, and the infinite list [0, 0, 0, . . .] in the λ-calculus, realising the recursion equations

$$\begin{aligned}
\mathbf{fact}\,N &= \mathbf{if}\,(\mathbf{iszero}\,N)\,\underline{1}\,(\mathbf{mult}\,N\,(\mathbf{fact}\,(\mathbf{pre}\,N))) \\
\mathbf{append}\,Z\,W &= \mathbf{if}\,(\mathbf{null}\,Z)\,W\,(\mathbf{cons}\,(\mathbf{hd}\,Z)\,(\mathbf{append}\,(\mathbf{tl}\,Z)\,W)) \\
\mathbf{zeroes} &= \mathbf{cons}\,\underline{0}\,\mathbf{zeroes}
\end{aligned}$$

To realize these, we simply put

$$\begin{aligned}
\mathbf{fact} &\equiv \mathbf{Y}\,(\lambda g n.\,\mathbf{if}\,(\mathbf{iszero}\,n)\,\underline{1}\,(\mathbf{mult}\,n(g(\mathbf{pre}\,n)))) \\
\mathbf{append} &\equiv \mathbf{Y}\,(\lambda g z w.\,\mathbf{if}\,(\mathbf{null}\,z)\,w\,(\mathbf{cons}\,(\mathbf{hd}\,z)\,(g(\mathbf{tl}\,z)\,w))) \\
\mathbf{zeroes} &\equiv \mathbf{Y}\,(\lambda g.\,\mathbf{cons}\,\underline{0}\,g)
\end{aligned}$$

In each definition, the recursive call is replaced by the variable g in $\mathbf{Y}\,(\lambda g. \ldots)$. Let us verify the recursion equation for **zeroes**; the others are similar:

$$\begin{aligned}
\mathbf{zeroes} &\equiv \mathbf{Y}\,(\lambda g.\,\mathbf{cons}\,\underline{0}\,g) \\
&= (\lambda g.\,\mathbf{cons}\,\underline{0}\,g)\,(\mathbf{Y}\,(\lambda g.\,\mathbf{cons}\,\underline{0}\,g)) \\
&= (\lambda g.\,\mathbf{cons}\,\underline{0}\,g)\,\mathbf{zeroes} \\
&\to \mathbf{cons}\,\underline{0}\,\mathbf{zeroes}
\end{aligned}$$

4.3 Usage of Y

In general, the recursion equation $M = PM$, where P is any λ-term, is satisfied by defining $M \equiv YP$. Let us consider the special case where M is to be an n-argument function. The equation $Mx_1 \ldots x_n = PM$ is satisfied by defining

$$M \equiv \mathbf{Y}\,(\lambda g x_1 \ldots x_n . Pg)$$

for then

$$\begin{array}{rcl} Mx_1 \ldots x_n & \equiv & \mathbf{Y}\,(\lambda g x_1 \ldots x_n . Pg) x_1 \ldots x_n \\ & = & (\lambda g x_1 \ldots x_n . Pg) M x_1 \ldots x_n \\ & \twoheadrightarrow & PM \end{array}$$

Let us realize the mutual recursive definition of M and N, with corresponding bodies P and Q:

$$\begin{array}{rcl} M & = & PMN \\ N & = & QMN \end{array}$$

The idea is to take the fixed point of a function F on pairs, such that $F(X, Y) = (PXY,\ QXY)$. Using our encoding of pairs, define

$$\begin{array}{rcl} L & \equiv & \mathbf{Y}\,(\lambda z.\, \mathbf{pair}\,(P(\mathbf{fst}\,z)(\mathbf{snd}\,z)) \\ & & \qquad\qquad (Q(\mathbf{fst}\,z)(\mathbf{snd}\,z))) \\ M & \equiv & \mathbf{fst}\,L \\ N & \equiv & \mathbf{snd}\,L \end{array}$$

By the fixed point property,

$$\begin{array}{l} L = \mathbf{pair}\,(P(\mathbf{fst}\,L)(\mathbf{snd}\,L)) \\ \qquad\qquad (Q(\mathbf{fst}\,L)(\mathbf{snd}\,L)) \end{array}$$

and by applying projections, we obtain the desired

$$M = P(\mathbf{fst}\,L)(\mathbf{snd}\,L) = PMN$$

$$N = Q(\mathbf{fst}\,L)(\mathbf{snd}\,L) = QMN.$$

4.4 Defining Fixed Point Combinators

The combinator $\mathbf{Y}$ was discovered by Haskell B. Curry. It is defined by

$$\mathbf{Y} \equiv \lambda f.(\lambda x.\, f(xx))(\lambda x.\, f(xx))$$

Let us calculate to show the fixed point property:

$$\begin{array}{rcl} \mathbf{Y}\,F & \to & (\lambda x.F(xx))(\lambda x.F(xx)) \\ & \to & F((\lambda x.F(xx))(\lambda x.F(xx))) \\ & = & F(\mathbf{Y}\,F) \end{array}$$

This consists of two β-reductions followed by a β-expansion. No reduction $\mathbf{Y}\,F \twoheadrightarrow F(\mathbf{Y}\,F)$ is possible! There are other fixed point combinators, such as Alan Turing's Θ:

$$\begin{array}{rcl} A & \equiv & \lambda xy.y(xxy) \\ \Theta & \equiv & AA \end{array}$$

We indeed have the reduction $\Theta F \twoheadrightarrow F(\Theta F)$:

$$\Theta F \equiv AAF \twoheadrightarrow F(AAF) \equiv F(\Theta F)$$

Here is a fixed point combinator discovered by Klop:

$$\begin{array}{rcl} \pounds & \equiv & \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator) \\ \$ & \equiv & \pounds \end{array}$$

The proof of $\$F \twoheadrightarrow F(\$F)$ is left as an exercise. Hint: look at the occurrences of r!

Any fixed point combinator can be used to make recursive definitions under call-by-name reduction. Later, we shall modify $\mathbf{Y}$ to get a fixed point combinator that works with a call-by-value interpreter for the λ-calculus. In practical compilers, recursion should be implemented directly because fixed point combinators are inefficient.

4.5 Head Normal Form

If $M = xM$ then M has no normal form. For if $M \twoheadrightarrow N$ where N is in normal form, then $N = xN$. Since xN is also in normal form, the Church-Rosser Theorem gives us $N \equiv xN$. But clearly N cannot contain itself as a subterm!

By similar reasoning, if $M = PM$ then M usually has no normal form, unless P is something like a constant function or identity function. So anything defined with the help of a fixed point combinator, such as **fact**, is unlikely to have a normal form.

Although **fact** has no normal form, we can still compute with it; **fact** $\underline{5}$ does have a normal form, namely $\underline{120}$. We can use infinite objects (including functions as above, and also lazy lists) by computing with them for a finite time and requesting a finite part of the result. To formalize this practice, let us define the notion of *head normal form* (hnf).

Definition 7 A term is in *head normal form* (hnf) if and only if it looks like this:

$$\lambda x_1 \ldots x_m . y M_1 \ldots M_k \qquad (m,\ k \geq 0)$$

Examples of terms in hnf include

$$x \qquad \lambda x.y\Omega \qquad \lambda x\, y.x \qquad \lambda z.z((\lambda x.a)c)$$

But $\lambda y.(\lambda x.a)y$ is not in hnf because it admits the so-called *head reduction*

$$\lambda y.(\lambda x.a)y \to \lambda y.a.$$

Let us note some obvious facts. A term in normal form is also in head normal form. Furthermore, if

$$\lambda x_1 \ldots x_m . y M_1 \ldots M_k \twoheadrightarrow N$$

then N *must* have the form

$$\lambda x_1 \ldots x_m . y N_1 \ldots N_k$$

where $M_1 \twoheadrightarrow N_1, \ldots, M_k \twoheadrightarrow N_k$. Thus, a head normal form fixes the outer structure of any further reductions and the final normal form (if any!). And since the arguments $M_1, \ldots, M_k$ cannot interfere with one another, they can be evaluated independently.

By reducing a term M to hnf we obtain a finite amount of information about the value of M. By further computing the hnfs of $M_1, \ldots, M_k$ we obtain the next layer of this value. We can continue evaluating to any depth and can stop at any time.

For example, define $\mathbf{ze} \equiv \Theta(\,\mathbf{pair}\ \underline{0}\,)$. This is analogous to **zeroes**, but uses pairs: $\mathbf{ze} = (0, (0, (0, \ldots)))$. We have

$$\begin{aligned}
\mathbf{ze} &\twoheadrightarrow \mathbf{pair}\ \underline{0}\,\mathbf{ze} \\
&\equiv (\lambda xyf.fxy)\,\underline{0}\,\mathbf{ze} \\
&\twoheadrightarrow \lambda f.f\ \underline{0}\,\mathbf{ze} \\
&\twoheadrightarrow \lambda f.f\ \underline{0}\,(\lambda f.f\ \underline{0}\,\mathbf{ze}) \\
&\twoheadrightarrow \ldots
\end{aligned}$$

With $\lambda f.f\ \underline{0}\,\mathbf{ze}$ we reached a head normal form, which we continued to reduce. We have $\mathbf{fst}\,(\mathbf{ze}) \twoheadrightarrow \underline{0}$ and $\mathbf{fst}\,(\,\mathbf{snd}\,(\mathbf{ze})) \twoheadrightarrow \underline{0}$, since the same reductions work if **ze** is a function's argument. These are examples of useful finite computations involving an infinite value.

Some terms do not even have a head normal form. Recall Ω, defined by $\Omega = (\lambda x.xx)(\lambda x.xx)$. A term is reduced to hnf by repeatedly performing leftmost reductions. With Ω we can only do $\Omega \to \Omega$, which makes no progress towards an hnf. Another term that lacks an hnf is $\lambda y.\Omega$; we can only reduce $\lambda y.\Omega \to \lambda y.\Omega$.

It can be shown that if MN has an hnf then so does M. Therefore, if M has no hnf then neither does any term of the form $MN_1N_2 \ldots N_k$. A term with no hnf behaves like a *totally undefined function*: no matter what you supply as arguments, evaluation never returns any information. It is not hard to see that if M has no hnf then neither does $\lambda x.M$ or $M[N/x]$, so M really behaves like a black hole. The only way to get rid of M is by a reduction such as $(\lambda x.a)M \to a$. This motivates the following definition.

Definition 8 A term is *defined* if and only if it can be reduced to head normal form; otherwise it is *undefined*.

The exercises below, some of which are difficult, explore this concept more deeply.

Exercise 6 Are the following terms defined? (Here $\mathbf{K} \equiv \lambda xy.x$.)

$$\mathbf{Y} \quad \mathbf{Y}\,\mathbf{not} \quad \mathbf{K} \quad \mathbf{Y}\,\mathbf{I} \quad x\Omega \quad \mathbf{Y}\,\mathbf{K} \quad \mathbf{Y}\,(\mathbf{K}\,x) \quad \underline{n}$$

Exercise 7 A term M is called *solvable* if and only if there exist variables $x_1, \ldots, x_m$ and terms $N_1, \ldots, N_n$ such that

$$(\lambda x_1 \ldots x_m.M)N_1 \ldots N_n = \mathbf{I}\,.$$

Investigate whether the terms given in the previous exercise are solvable.

Exercise 8 Show that if M has an hnf then M is solvable. Wadsworth proved that M is solvable if and only if M has an hnf, but the other direction of the equivalence is much harder.

4.6 Aside: An Explanation of Y

For the purpose of expressing recursion, we may simply exploit $\mathbf{Y}\,F = F(\mathbf{Y}\,F)$ without asking why it holds. However, the origins of $\mathbf{Y}$ have interesting connections with the development of mathematical logic.

Alonzo Church invented the λ-calculus to formalize a new set theory. Bertrand Russell had (much earlier) demonstrated the inconsistency of naive set theory. If

we are allowed to construct the set $R \equiv \{x \mid x \notin x\}$, then $R \in R$ if and only if $R \notin R$. This became known as Russell's Paradox.

In his theory, Church encoded sets by their characteristic functions (equivalently, as predicates). The membership test $M \in N$ was coded by the application $N(M)$, which might be true or false. The set abstraction $\{x \mid P\}$ was coded by $\lambda x.P$, where P was some λ-term expressing a property of x.

Unfortunately for Church, Russell's Paradox was derivable in his system! The Russell set is encoded by $R \equiv \lambda x.\,\mathbf{not}\,(xx)$. This implied $RR = \mathbf{not}\,(RR)$, which was a contradiction if viewed as a logical formula. In fact, RR has no head normal form: it is an undefined term like Ω.

Curry discovered this contradiction. The fixed point equation for $\mathbf{Y}$ follows from $RR = \mathbf{not}\,(RR)$ if we replace **not** by an arbitrary term F. Therefore, $\mathbf{Y}$ is often called the Paradoxical Combinator.

Because of the paradox, the λ-calculus survives only as an equational theory. The *typed* λ-calculus does not admit any known paradoxes and is used to formalize the syntax of higher-order logic.

4.7 Summary: the λ-Calculus Versus Turing Machines

The λ-calculus can encode the common data structures, such as booleans and lists, such that they satisfy natural laws. The λ-calculus can also express recursive definitions. Because the encodings are technical, they may appear to be unworthy of study, but this is not so.

- The encoding of the natural numbers via Church numerals is valuable in more advanced calculi, such as the second-order λ-calculus.
- The encoding of lists via ordered pairs models their usual implementation on the computer.
- As just discussed, the definition of $\mathbf{Y}$ formalizes Russell's Paradox.
- Understanding recursive definitions as fixed points is the usual treatment in semantic theory.

These constructions and concepts are encountered throughout theoretical computer science. That cannot be said of any Turing machine program!

5 The λ-Calculus and Computation Theory

The λ-calculus is one of the classical models of computation, along with Turing machines and general recursive functions. *Church's Thesis* states that the computable functions are precisely those that are λ-definable. Below, we shall see that

the λ-calculus has the same power as the (total) recursive functions. We shall also see some strong undecidability results for λ-terms. The following definition is fundamental.

Definition 9 If f is an n-place function over the natural numbers, then f is λ-*definable* if there exists some λ-term F such that, for all $k_1, \ldots, k_n \in N$,

$$F\,\underline{k_1}\,\ldots\,\underline{k_n} = \underline{f(k_1, \ldots, k_n)}\,.$$

In other words, F maps numerals for arguments to numerals for f's results. By the Church-Rosser Theorem, since numerals are in normal form, we have the reduction

$$F\,\underline{k_1}\,\ldots\,\underline{k_n} \twoheadrightarrow \underline{f(k_1, \ldots, k_n)}\,.$$

Thus, we can compute the value of $f(k_1, \ldots, k_n)$ by normalizing the term $F\,\underline{k_1}\,\ldots\,\underline{k_n}$.

5.1 The Primitive Recursive Functions

In computation theory, the primitive recursive functions are built up from the following basic functions:

0	the constant zero
suc	the successor function
U_n^i	the projection functions, $U_n^i(x_1, \ldots, x_n) = x_i$

New functions are constructed by *substitution* and *primitive recursion*. Substitution, or generalized composition, takes an m-place function g and the n-place functions $h_1, \ldots, h_m$; it yields the n-place function f such that

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)).$$

Primitive recursion takes an n-place function g and an $(n+2)$-place function h; it yields the $(n+1)$-place function f such that

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(\mathbf{suc}(y), x_1, \ldots, x_n) &= h(f(y, x_1, \ldots, x_n), y, x_1, \ldots, x_n) \end{aligned}$$

5.2 Representing the Primitive Recursive Functions

The functions are defined in the λ-calculus in the obvious way. The proof that they are correct (with respect to our definition of λ-definability) is left as an exercise.

Here are the basic functions:

- For 0 use $\underline{0}$, namely $\lambda f x.x$.
- For suc use **suc**, namely $\lambda n f x.nf(fx)$.
- For U_n^i use $\lambda x_1 \ldots x_n.x_i$.

To handle substitution, suppose that the m-place function g and n-place functions $h_1, \ldots, h_m$ are λ-defined by G, $H_1, \ldots, H_m$, respectively. Then, their composition f is λ-defined by

$$F \equiv \lambda x_1 \ldots x_n.G(H_1 x_1 \ldots x_n) \ldots (H_m x_1 \ldots x_n).$$

To handle primitive recursion, suppose that the n-place function g and $(n+2)$-place function h are λ-defined by G and H, respectively. The primitive recursive function f is λ-defined by

$$\begin{aligned} F \equiv\ \mathbf{Y}\Big(\lambda f y x_1 \ldots x_n.\ &\mathbf{if}\ (\mathbf{iszero}\ y) \\ &(G x_1 \ldots x_n) \\ &(H(f(\mathbf{pre}\ y)x_1 \ldots x_n)(\mathbf{pre}\ y)x_1 \ldots x_n))\Big). \end{aligned}$$

5.3 The General Recursive Functions

Starting with the primitive recursive functions, the so-called *general recursive functions* are obtained by adding the *minimisation operator*, or function inversion. Given an n-place function g it yields the n-place function f such that

$$f(x_1, \ldots, x_n) = \text{the least } y \text{ such that } [g(y, x_2, \ldots, x_n) = x_1]$$

and is undefined if no such y exists.

Thus, minimisation may yield a partial function. This leads to certain difficulties.

- The notion of *undefined* is complicated in the λ-calculus. It is tedious to show that the λ-encoding of minimisation yields an undefined term if and only if the corresponding function fails to terminate.
- Composition in the λ-calculus is non-strict. For example, consider the partial functions r and s such that $r(x) = 0$ and $s(x)$ is undefined for all x. We may λ-define r and s by $R \equiv \lambda x.\underline{0}$ and $S \equiv \lambda x.\Omega$. Now, $r(s(0))$ should be undefined, but $R(S\,\underline{0}) \to \underline{0}$. Defining the strict composition of R and S is tricky.

Let us restrict attention to the *total* recursive functions. If for all $x_1, \ldots, x_n$ there is some y such that $g(y, x_2, \ldots, x_n) = x_1$, then the inverse of g is total. Suppose that G is a term that λ-defines g. Then the inverse is λ-defined by

$$F \equiv \lambda x_1 \ldots x_n.\, \mathbf{Y} \Big(\lambda h y.\, \mathbf{if}\, (\mathbf{equals}\, x_1 (G y x_2 \ldots x_n) \; y \; (h(\mathbf{suc}\, y)) \Big)\; \underline{0}$$

This sets up a recursive function that tests whether $(G y x_2 \ldots x_n)$ equals x_1 for increasing values of y. To start the search, this function is applied to $\underline{0}$. The equality test for natural numbers can be defined in many ways, such as

$$\mathbf{equals} \equiv \lambda m\, n.\, \mathbf{iszero}\, (\, \mathbf{add}\, (\mathbf{sub}\, m\, n)(\mathbf{sub}\, n\, m))$$

This works because $\mathbf{sub}\, m n = \underline{0}$ if $m \leq n$. The equality relation between arbitrary λ-terms (not just for Church numerals) is undecidable and cannot be expressed in the λ-calculus.

Exercise 9 Find another definition of the equality test on the natural numbers.

5.4 The λ-Definable Functions are Recursive

We have just shown that all total recursive functions are λ-definable. The converse is not particularly difficult and we only sketch it. The key is to assign a unique (and recursively computable!) Gödel number $\#M$ to each λ-term M. First, assume that the set of variables has the form $x_1, x_2, x_3, \ldots$, so that each variable is numbered. The definition of $\#M$ is quite arbitrary; for example, we could put

$$\begin{aligned} \#x_i &= 2^i \\ \#(\lambda x_i.M) &= 3^i 5^{\#M} \\ \#(MN) &= 7^{\#M} 11^{\#N} \end{aligned}$$

To show that all λ-definable functions are recursive, suppose that we are given a λ-term F; we must find a recursive function f such that $f(k_1, \ldots, k_n) = k$ if and only if $F\, \underline{k_1} \ldots \underline{k_n} = \underline{k}$. We can do this in a uniform fashion by writing an interpreter for the λ-calculus using the language of total recursive functions, operating upon the Gödel numbers of λ-terms. This is simply a matter of programming. The resulting program is fairly long but much easier to understand than its Turing machine counterpart, namely the Universal Turing Machine.

Exercise 10 (Composition of partial functions.) Show that $\underline{n}\,\mathbf{I} = \mathbf{I}$ for every Church numeral $\underline{n}$. Use this fact to show that the term H defined by

$$H \equiv \lambda x.G\,x\,\mathbf{I}\,(F(G\,x))$$

λ-defines the composition $f \circ g$ of the functions f and g, provided F λ-defines f and G λ-defines g. What can be said about $H\,M$ if $G\,M$ is undefined? Hint: recall the definition of *solvable* from Exercise 7.

5.5 The Second Fixed Point Theorem

We shall formalize the λ-calculus in itself in order to prove some undecidability results. The Gödel numbering is easily coded and decoded by arithmetic operations. In particular, there exist computable functions Ap and Num, operating on natural numbers, such that

$$\begin{aligned} \mathrm{Ap}(\#M, \#N) &= \#(MN) \\ &= 7^{\#M}11^{\#N} \\ \mathrm{Num}(n) &= \#(\underline{n}) \\ &= \#\lambda x_2\,x_1.\underbrace{x_2(\cdots(x_2}_{n}\,x_1)\cdots) \end{aligned}$$

These can be λ-defined by terms **AP** and **NUM**, operating on Church numerals. Let us write $\ulcorner M \urcorner$ for $\underline{(\#M)}$, which is the Church numeral for the Gödel number of M. This itself is a λ-term. Using this notation, **AP** and **NUM** satisfy

$$\begin{aligned} \mathbf{AP}\ulcorner M\urcorner\ulcorner N\urcorner &= \ulcorner MN\urcorner \\ \mathbf{NUM}\,\underline{n} &= \ulcorner \underline{n} \urcorner \qquad \text{(where } \underline{n} \text{ is any Church numeral)} \end{aligned}$$

Putting $\ulcorner M \urcorner$ for $\underline{n}$ in the latter equation yields a crucial property:

$$\mathbf{NUM}\ulcorner M\urcorner = \ulcorner\ulcorner M\urcorner\urcorner.$$

Theorem 10 (Second Fixed Point) *If F is any λ-term then there exists a term X such that*

$$F\ulcorner X\urcorner = X.$$

Proof Make the following definitions:

$$\begin{aligned} W &\equiv \lambda x.F(\mathbf{AP}\,x(\mathbf{NUM}\,x)) \\ X &\equiv W\ulcorner W\urcorner \end{aligned}$$

Then, by a β-reduction and by the laws for **AP** and **NUM**, we get

$$\begin{aligned} X &\rightarrow F(\mathbf{AP}\ulcorner W\urcorner(\mathbf{NUM}\ulcorner W\urcorner)) \\ &\twoheadrightarrow F(\mathbf{AP}\ulcorner W\urcorner\ulcorner\ulcorner W\urcorner\urcorner) \\ &\twoheadrightarrow F(\ulcorner W\ulcorner W\urcorner\urcorner) \\ &\equiv F(\ulcorner X\urcorner) \end{aligned}$$

Therefore $X = F(\ulcorner X\urcorner)$. □

Exercise 11 How do we know that the steps in the proof are actually reductions, rather than mere equalities?

What was the First Fixed Point Theorem? If F is any λ-term then there exists a term X such that $FX = X$. Proof: put $X \equiv \mathbf{Y}F$. Note the similarity between $\mathbf{Y}F$ and the X constructed above. We use fixed point combinators to λ-define recursive functions; we shall use the Second Fixed Point Theorem to prove undecidability of the halting problem.

5.6 Undecidability Results for the λ-Calculus

We can show that the halting problem, as expressed in the λ-calculus, is undecidable.

Theorem 11 *There is no* λ*-term* **halts** *such that*

$$\mathbf{halts}\ulcorner M\urcorner = \begin{cases} \mathbf{true} & \textit{if } M \textit{ has a normal form} \\ \mathbf{false} & \textit{if } M \textit{ has no normal form} \end{cases}$$

Proof Assume the contrary, namely that **halts** exists. Put

$$D \equiv \lambda x.\,\mathbf{if}\,(\mathbf{halts}\,x)\Omega\,\underline{0}\,;$$

by the Second Fixed Point Theorem, there exists X such that

$$X = D\ulcorner X\urcorner = \mathbf{if}\,(\mathbf{halts}\ulcorner X\urcorner)\Omega\,\underline{0}\,.$$

There are two possible cases; both lead to a contradiction:

- If X has a normal form then $X = \mathbf{if}\,\mathbf{true}\,\Omega\,\underline{0} = \Omega$, which has no normal form!
- If X has no normal form then $X = \mathbf{if}\,\mathbf{false}\,\Omega\,\underline{0} = \underline{0}$, which does have a normal form!

□

The proof is a typical diagonalisation argument. The theorem is strong: although **halts**$\ulcorner M \urcorner$ can do anything it likes with the code of M, analysing the term's structure, it cannot determine whether M has a normal form. Assuming Church's Thesis — the computable functions are precisely the λ-definable ones — the theorem states that the halting problem for the normalization of λ-terms is computationally undecidable.

Much stronger results are provable. Dana Scott has shown that

- if $\mathcal{A}$ is *any* non-trivial set of λ-terms (which means that $\mathcal{A}$ is neither empty nor the set of all λ-terms), and
- if $\mathcal{A}$ is closed under equality (which means that $M \in \mathcal{A}$ and $M = N$ imply $N \in \mathcal{A}$)

then the test for membership in A is undecidable. The halting problem follows as a special case, taking

$$\mathcal{A} = \{M \mid M = N \text{ and } N \text{ is in normal form}\}$$

See Barendregt [1, page 143], for more information.

6 ISWIM: The λ-calculus as a Programming Language

Peter Landin was one of the first computer scientists to take notice of the λ-calculus and relate it to programming languages. He observed that Algol 60's scope rules and call-by-name rule had counterparts in the λ-calculus. In his paper [8], he outlined a skeletal programming languages based on the λ-calculus. The title referred to the 700 languages said to be already in existence; in principle, they could all share the same λ-calculus skeleton, differing only in their data types and operations. Landin's language, ISWIM (If you See What I Mean), dominated the early literature on functional programming, and was the model for ML.

Lisp also takes inspiration from the λ-calculus, and appeared many years before ISWIM. But Lisp made several fatal mistakes: dynamic variable scoping, an imperative orientation, and no higher-order functions. Although ISWIM allows imperative features, Lisp is essentially an imperative language, because all variables may be updated.

ISWIM was designed to be extended with application-specific data and operations. It consisted of the λ-calculus plus a few additional constructs, and could be translated back into the pure λ-calculus. Landin called the extra constructs *syntactic sugar* because they made the λ-calculus more palatable.

6.1 Overview of ISWIM

ISWIM started with the λ-calculus:

x	variable
$(\lambda x.M)$	abstraction
(MN)	application

It also allowed local declarations:

let $x = M$ **in** N	simple declaration
let $f\ x_1 \cdots x_k = M$ **in** N	function declaration
letrec $f\ x_1 \cdots x_k = M$ **in** N	recursive declaration

Local declarations could be post-hoc:

N **where** $x = M$
N **where** $f\ x_1 \cdots x_k = M$
N **whererec** $f\ x_1 \cdots x_k = M$

The meanings of local declarations should be obvious. They can be translated into the pure λ-calculus:

let $x = M$ **in** N	$\equiv (\lambda x.N)M$
let $f\ x_1 \cdots x_k = M$ **in** N	$\equiv (\lambda f.N)(\lambda x_1 \cdots x_k.M)$
letrec $f\ x_1 \cdots x_k = M$ **in** N	$\equiv (\lambda f.N)(\mathbf{Y}\ (\lambda f x_1 \cdots x_k.M))$

Programmers were not expected to encode data using Church numerals and the like. ISWIM provided primitive data structures: integers, booleans and ordered pairs. There being no type system, lists could be constructed by repeated pairing, as in Lisp. The constants included

$0\ 1\ -1\ 2\ -2\ \ldots$	integers
$+\ -\ \times\ /$	arithmetic operators
$=\ \neq\ <\ >\ \leq\ \geq$	relational operators
true false	booleans
and or not	boolean connectives
if E **then** M **else** N	conditional

6.2 Call-by-value in ISWIM

The *call-by-value* rule, rather than *call-by-name*, was usually adopted. This was (and still is) easier to implement; we shall shortly see how this was done, using the SECD machine. Call-by-value is indispensable in the presence of imperative operations.

Call-by-value gives more intuitive and predictable behaviour generally. Classical mathematics is based on strict functions; an expression is undefined unless all its parts are defined. Under call-by-name we can define a function f such that if $f(x) = 0$ for all x, with even $f(1/0) = 0$. Ordinary mathematics cannot cope with such functions; putting them on a rigorous basis requires complex theories.

Under call-by-value, *if-then-else* must be taken as a special form of expression. Treating **if** as a function makes *fact* run forever:

$$\mathit{letrec}\ \mathit{fact}(n) = \mathbf{if}\ (n = 0)\ 1\ (n \times \mathit{fact}(n-1))$$

The arguments to **if** are always evaluated, including the recursive call; when $n = 0$ it tries to compute $\mathit{fact}(-1)$. Therefore, we take conditional expressions as primitive, with evaluation rules that return M or N *unevaluated*:

$$\begin{array}{lcl} \mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N & \to & M \\ \mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N & \to & N \end{array}$$

Our call-by-value rule never reduces anything enclosed by a λ. So we can translate the conditional expression to the application of an **if**-function:

$$\mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N \equiv \mathbf{if}\ E\ (\lambda u.M)\ (\lambda u.N)\ 0$$

Choosing some variable u not free in M or N, enclosing those expressions in λ delays their evaluation; finally, the selected one is applied to 0.

6.3 Pairs, Pattern-Matching and Mutual Recursion

ISWIM includes ordered pairs:

(M, N)	pair constructor
fst **snd**	projection functions

For pattern-matching, let $\lambda(p_1, p_2).E$ abbreviate

$$\lambda z.(\lambda p_1\ p_2.E)(\mathbf{fst}\, z)(\mathbf{snd}\, z)$$

where p_1 and p_2 may themselves be patterns. Thus, we may write

$\mathit{let}\ (x, y) = M\ \mathbf{in}\ E$	taking apart M's value
$\mathit{let}\ f(x, y) = E\ \mathbf{in}\ N$	defining f on pairs

The translation iterates to handle things like

$$\mathit{let}\ (w, (x, (y, z))) = M\ \mathbf{in}\ E.$$

We may introduce n-tuples, writing $(x_1, \ldots, x_{n-1}, x_n)$ for the nested pairs

$$(x_1, \ldots, (x_{n-1}, x_n) \ldots).$$

The mutually recursive function declaration

$$\begin{array}{ll} \mathit{letrec} & f_1\,\vec{x}_1 = M_1 \\ \mathit{and} & f_2\,\vec{x}_2 = M_2 \\ \vdots & \\ \mathit{and} & f_k\,\vec{x}_k = M_k \\ \mathit{in}\ N & \end{array}$$

can be translated to an expression involving pattern-matching:

$$(\lambda(f_1, \ldots, f_k).N)(\mathbf{Y}\,(\lambda(f_1, \ldots, f_k).(\lambda\vec{x}_1.M_1, \lambda\vec{x}_2.M_2, \ldots, \lambda\vec{x}_k.M_k)))$$

We can easily handle the general case of k mutually recursive functions, each with any number of arguments. Observe the power of syntactic sugar!

6.4 From ISWIM to ML

Practically all programming language features, including go to statements and pointer variables, can be formally specified in the λ-calculus, using the techniques of *denotational semantics*. ISWIM is much simpler than that; it is programming directly in the λ-calculus. To allow imperative programming, we can even define sequential execution, letting $M; N$ abbreviate $(\lambda x.N)M$; the call-by-value rule will evaluate M before N. However, imperative operations must be adopted as primitive; they cannot be defined by simple translation into the λ-calculus.

ISWIM gives us all the basic features of a programming language — variable scope rules, function declarations, and local declarations. (The *let* declaration is particularly convenient; many languages still make us write assignments for this purpose!) To get a real programming language, much more needs to be added, but the languages so obtained will have a common structure.

ISWIM was far ahead of its time and never found mainstream acceptance. Its influence on ML is obvious. Standard ML has changed the syntax of declarations, added polymorphic types, exceptions, fancier pattern-matching and modules — but much of the syntax is still defined by translation. A French dialect of ML, called CAML, retains much of the traditional ISWIM syntax [3].

6.5 The SECD Machine

Landin invented the SECD machine, an interpreter for the λ-calculus, in order to execute ISWIM programs [2, 4, 7]. A variant of the machine executes instructions

compiled from λ-terms. With a few optimisations, it can be used to implement real functional languages, such as ML. SECD machines can be realized as byte-code interpreters, their instructions can be translated to native code, and they can be implemented directly on silicon. The SECD machine yields strict evaluation, call-by-value. A lazy version is much slower than graph reduction of combinators, which we shall consider later.

It is tempting to say that a *value* is any fully evaluated λ-term, namely a term in normal form. This is a poor notion of value in functional programming, for two reasons:

1. Functions themselves should be values, but many functions have no normal form. Recursive functions, coded as $\mathbf{Y}\,F$, satisfy $\mathbf{Y}\,F = F(\mathbf{Y}\,F) = F(F(\mathbf{Y}\,F)) = \cdots$. Although they have no normal form, they may well yield normal forms as results when they are applied to arguments.

2. Evaluating the body of a λ-abstraction, namely the M in $\lambda x.M$, serve little purpose; we are seldom interested in the internal structure of a function. Only when it is applied to some argument N do we demand the result and evaluate $M[N/x]$.

Re (2), we clearly cannot use encodings like $\lambda x\,y.x$ for **true** and $\lambda f\,x.x$ for $\underline{0}$, since our evaluation rule will not reduce function bodies. We must take the integers, booleans, pairs, etc., as primitive constants. Their usual functions ($+$, $-$, $\times$, ...) must also be primitive constants.

6.6 Environments and Closures

Consider the reduction sequence

$$(\lambda xy.x + y)\,3\,5 \to (\lambda y.3 + y)\,5 \to 3 + 5 \to 8.$$

The β-reduction eliminates the free occurrence of x in $\lambda y.x + y$ by substitution for x. Substitution is too slow to be effective for parameter passing; instead, the SECD machine records $x = 3$ in an *environment*.

With curried functions, $(\lambda xy.x + y)\,3$ is a legitimate value. The SECD machine represents it by a *closure*, packaging the λ-abstraction with its current environment:

$$\begin{array}{lccccc} \mathbf{Clo}(& y & , & x+y & , & x=3 \quad) \\ & \uparrow & & \uparrow & & \uparrow \\ & \text{bound variable} & & \text{function body} & & \text{environment} \end{array}$$

When the SECD machine applies this function value to the argument 5, it restores the environment to $x = 3$, adds the binding $y = 5$, and evaluates $x + y$ in this augmented environment.

A closure is so-called because it "closes up" the function body over its free variables. This operation is costly; most programming languages forbid using functions as values. Until recently, most versions of Lisp let a function's free variables pick up any values they happened to have in the environment of the call (not that of the function's definition!); with this approach, evaluating

$$\begin{array}{l} \mathit{let}\ g(y) = x + y\ \mathbf{in} \\ \mathit{let}\ f(x) = g(1)\ \ \mathbf{in} \\ f(17) \end{array}$$

would return 18, using 17 as the value of x in g! This is *dynamic binding*, as opposed to the usual *static binding*. Dynamic binding is confusing because the scope of x in $f(x)$ can extend far beyond the body of f — it includes all code reachable from f (including g in this case).

Common Lisp, now the dominant version, corrects this long-standing Lisp deficiency by adopting static binding as standard. It also allows dynamic binding, though.

6.7 The SECD State

The SECD machine has a state consisting of four components S, E, C, D:

1. The *Stack* is a list of values, typically operands or function arguments; it also returns the result of a function call.

2. The *Environment* has the form $x_1 = a_1; \cdots ; x_n = a_n$, expressing that the variables $x_1, \ldots, x_n$ have the values $a_1, \ldots, a_n$, respectively.

3. The *Control* is a list of commands. For the interpretive SECD machine, a command is a λ-term or the word **app**; the compiled SECD machine has many commands.

4. The *Dump* is empty $(-)$ or is another machine state of the form (S, E, C, D). A typical state looks like

   $$(S_1, E_1, C_1, (S_2, E_2, C_2, \ldots (S_n, E_n, C_n, -) \ldots))$$

   It is essentially a list of triples (S_1, E_1, C_1), (S_2, E_2, C_2), $\ldots$, (S_n, E_n, C_n) and serves as the function call stack.

6.8 State Transitions

Let us write SECD machine states as boxes:

Stack
Environment
Control
Dump

To evaluate the λ-term M, the machine begins execution an the *initial state* where M is the Control:

S	$-$
E	$-$
C	M
D	$-$

If the Control is non-empty, then its first command triggers a state transition. There are cases for constants, variables, abstractions, applications, and the **app** command.

A constant is pushed on to the Stack:

S		$k; S$
E	$\longmapsto$	E
$k; C$		C
D		D

The value of a variable is taken from the Environment and pushed on to the Stack. If the variable is x and E contains $x = a$ then a is pushed:

S		$a; S$
E	$\longmapsto$	E
$x; C$		C
D		D

A λ-abstraction is converted to a closure, then pushed on to the Stack. The closure contains the current Environment:

S		$\mathbf{Clo}(x, M, E); S$
E	$\longmapsto$	E
$\lambda x.M; C$		C
D		D

A function application is replaced by code to evaluate the argument and the function, with an explicit **app** instruction:

S
E
$MN; C$
D

$\longmapsto$

S
E
$N; M;$ **app**$; C$
D

The **app** command calls the function on top of the Stack, with the next Stack element as its argument. A primitive function, like $+$ or $\times$, delivers its result immediately:

$f; a; S$
E
app$; C$
D

$\longmapsto$

$f(a); S$
E
C
D

The closure **Clo**(x, M, E') is called by creating a new state to evaluate M in the Environment E', extended with a binding for the argument. The old state is saved in the Dump:

Clo$(x, M, E'); a; S$
E
app$; C$
D

$\longmapsto$

$-$
$x = a; E'$
M
(S, E, C, D)

The function call terminates in a state where the Control is empty but the Dump is not. To return from the function, the machine restores the state (S, E, C, D) from the Dump, then pushes a on to the Stack. This is the following state transition:

a
E'
$-$
(S, E, C, D)

$\longmapsto$

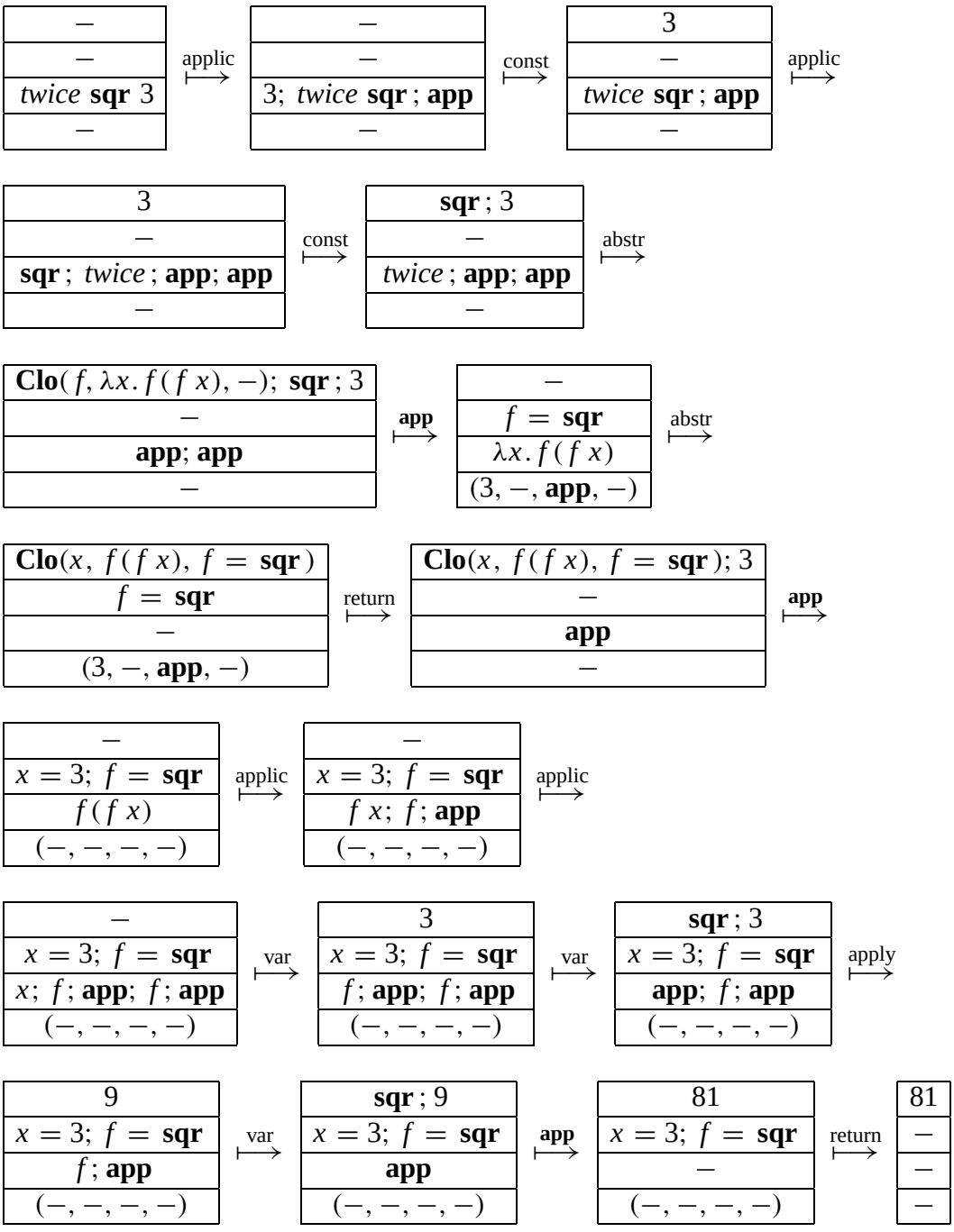
$a; S$
E
C
D

The result of the evaluation, say a, is obtained from a *final state* where the Control and Dump are empty, and a is the sole value on the Stack:

S	a
E	$-$
C	$-$
D	$-$

6.9 A Sample Evaluation

To demonstrate how the SECD machine works, let us evaluate the expression *twice* **sqr** 3, where *twice* is $\lambda f\, x.\, f\,(f\, x)$ and **sqr** is a built-in squaring function. (Note that *twice* is just the Church numeral $\underline{2}$). We start from the initial state:



The machine terminates in a final state, giving a value of 81.

6.10 The Compiled SECD Machine

It takes 17 steps to evaluate $((\lambda x\ y.x + y)\ 3)\ 5$! Much faster execution is obtained by first compiling the λ-term. Write $[\![M]\!]$ for the list of commands produced by compiling M; there are cases for each of the four kinds of λ-term.

Constants are compiled to the **const** command, which will (during later execution of the code) push a constant onto the Stack:

$$[\![k]\!] = \mathbf{const}(k)$$

Variables are compiled to the **var** command, which will push the variable's value, from the Environment, onto the Stack:

$$[\![x]\!] = \mathbf{var}(x)$$

Abstractions are compiled to the **closure** command, which will push a closure onto the Stack. The closure will include the current Environment and will hold M as a list of commands, from compilation:

$$[\![\lambda x.M]\!] = \mathbf{closure}(x, [\![M]\!])$$

Applications are compiled to the **app** command at compile time. Under the interpreted SECD machine, this work occurred at run time:

$$[\![MN]\!] = [\![N]\!]; [\![M]\!]; \mathbf{app}$$

We could add further instructions, say for *conditionals*. Let $\mathbf{test}(C_1, C_2)$ be replaced by C_1 or C_2, depending upon whether the value on top of the Stack is **true** or **false**:

$$[\![\ \mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N]\!] = [\![E]\!]; \mathbf{test}([\![M]\!], [\![N]\!])$$

To allow built-in 2-place functions such as $+$ and $\times$ could be done in several ways. Those functions could be made to operate upon ordered pairs, constructed using a **pair** instruction. More efficient is to introduce arithmetic instructions such as **add** and **mult**, which pop both their operands from the Stack. Now $((\lambda x\ y.x + y)\ 3)\ 5$ compiles to

$$\mathbf{const}(5); \mathbf{const}(3); \mathbf{closure}(x, C_0); \mathbf{app}; \mathbf{app}$$

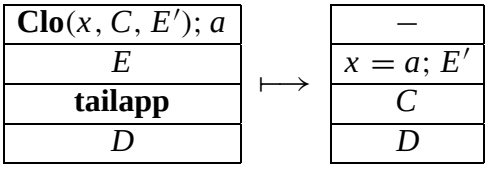
and generates two further lists of commands:

$$\begin{aligned} C_0 &= \mathbf{closure}(y, C_1) \\ C_1 &= \mathbf{var}(y); \mathbf{var}(x); \mathbf{add} \end{aligned}$$

Many further optimisations can be made, leading to an execution model quite close to conventional hardware. Variable names could be removed from the Environment, and bound variables referred to by depth rather than by name. Special instructions **enter** and **exit** could efficiently handle functions that are called immediately (say, those created by the declaration **let**$x = N$**in**M), creating no closure:

$$[\![(\lambda x.M)N]\!] = [\![N]\!]; \textbf{enter}; [\![M]\!]; \textbf{exit}$$

Tail recursive (sometimes called *iterative*) function calls could be compiled to the **tailapp** command, which would cause the following state transition:



The useless state $(-, E, -, D)$ is never stored on the dump, and the function return after **tailapp** is never executed — the machine jumps directly to C!

6.11 Recursive Functions

The usual fixed point combinator, $\mathbf{Y}$, fails under the SECD machine; it always loops. A modified fixed point combinator, including extra λ's to delay evaluation, does work:

$$\lambda f.(\lambda x.\, f(\lambda y.x\ x\ y)(\lambda y.x\ x\ y))$$

But it is hopelessly slow! Recursive functions are best implemented by creating a closure with a pointer back to itself.

Suppose that $f(x) = M$ is a recursive function definition. The value of f is represented by $\mathbf{Y}\,(\lambda f\ x.M)$. The SECD machine should interpret $\mathbf{Y}\,(\lambda f x.M)$ in a special manner, applying the closure for $\lambda f\ x.M$ to a dummy value, $\perp$. If the current Environment is E then this yields the closure

$$\textbf{Clo}(x,\ M,\ f = \perp; E)$$

Then the machine modifies the closure, replacing the $\perp$ by a pointer looping back to the closure itself:

$$\textbf{Clo}(x,\ M,\ f = \cdot; E)$$

When the closure is applied, recursive calls to f in M will re-apply the same closure. The cyclic environment supports recursion efficiently.

The technique is called "tying the knot" and works only for function definitions. It does not work for recursive definitions of data structures, such as the infinite list $[0, 0, 0, \ldots]$, defined as $\mathbf{Y}\,(\lambda l.\,\mathbf{cons}\;0\,l)$. Therefore strict languages like ML allow only functions to be recursive.

7 Lazy Evaluation via Combinators

The SECD machine employs call-by-value. It can be modified for call-by-need (lazy evaluation), as follows. When a function is called, its argument is stored *unevaluated* in a closure containing the current environment. Thus, the call MN is treated something like $M(\lambda u.N)$, where u does not appear in N. This closure is called a *suspension*. When a strict, built-in function is called, such as $+$, its argument is evaluated in the usual way.

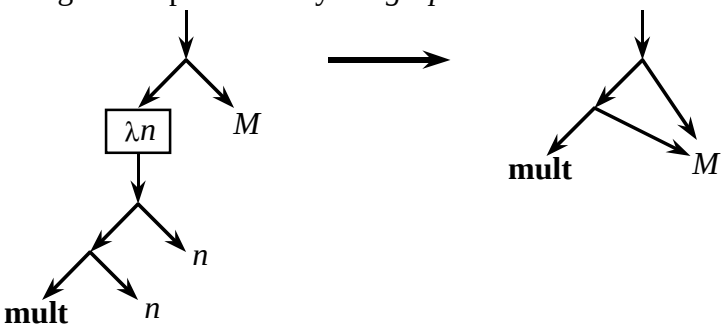
It is essential that no argument be evaluated more than once, no matter how many times it appears in the function's body:

$$\begin{array}{l} \mathit{let}\ sqr\ n = n \times n\ \mathbf{in}\ N \\ sqr\,(sqr\,(sqr\ 2)) \end{array}$$

If this expression were evaluated by repeatedly duplicating the argument of sqr, the waste would be intolerable. Therefore, the lazy SECD machine updates the environment with the value of the argument, after it is evaluated for the first time. But the cost of creating suspensions makes this machine ten times slower than the strict SECD machine, according to David Turner, and compilation can give little improvement.

7.1 Graph Reduction in the λ-Calculus

Another idea is to work directly with λ-terms, using sharing and updating to ensure that no argument is evaluated more than once. For instance, the evaluation of $(\lambda n.n \times n)M$ might be represented by the *graph reduction*



The difficulty here is that λ-abstractions may themselves be shared. We may not modify the body of the abstraction, replacing the bound variable by the actual argument. Instead, we must copy the body — including parts containing no occurrence of the bound variable — when performing the substitution.

Both the lazy SECD machine and graph reduction of λ-terms suffer because of the treatment of bound variables. Combinators have the same expressive power as the λ-calculus, but no bound variables. Graph reduction in combinators does not require copying. David Turner found an efficient method of translating λ-terms into combinators, for evaluation by graph reduction [9]. Offshoots of his methods have been widely adopted for implementing lazy functional languages.

7.2 Introduction to Combinators

In the simplest version, there are only two combinators, $\mathbf{K}$ and $\mathbf{S}$. Combinators are essentially constants. It is possible to define $\mathbf{K}$ and $\mathbf{S}$ in the λ-calculus, but combinatory logic (CL) exists as a theory in its own right.[2]

The terms of combinatory logic, written P, Q, R, ..., are built from $\mathbf{K}$ and $\mathbf{S}$ using application. They may contain free variables, but no bound variables. A typical CL term is $\mathbf{K}\,x\,(\mathbf{S}\,\mathbf{K}\,x)\,(\mathbf{K}\,\mathbf{S}\,\mathbf{K}\,y)\,\mathbf{S}$. Although CL is not particularly readable, it is powerful enough to code all the computable functions!

The combinators obey the following reductions:

$$\begin{array}{lcl} \mathbf{K}\,P\,Q & \to_w & P \\ \mathbf{S}\,P\,Q\,R & \to_w & P\,R\,(Q\,R) \end{array}$$

Thus, the combinators could have been defined in the λ-calculus by

$$\begin{array}{rcl} \mathbf{K} & \equiv & \lambda x\,y.x \\ \mathbf{S} & \equiv & \lambda x\,y\,z.x\,z\,(y\,z) \end{array}$$

But note that $\mathbf{S}\,\mathbf{K}$ does not reduce — because $\mathbf{S}$ requires three arguments — while the corresponding λ-term does. For this reason, combinator reduction is known as *weak reduction* (hence the "w" in $\to_w$).

Here is an example of weak reduction:

$$\mathbf{S}\,\mathbf{K}\,\mathbf{K}\,P \to_w \mathbf{K}\,P\,(\mathbf{K}\,P) \to_w P$$

Thus $\mathbf{S}\,\mathbf{K}\,\mathbf{K}\,P \twoheadrightarrow_w P$ for all combinator terms P; let us define the identity combinator by $\mathbf{I} \equiv \mathbf{S}\,\mathbf{K}\,\mathbf{K}$.

Many of the concepts of the λ-calculus carry over to combinators. A combinator term P is in *normal form* if it admits no weak reductions. Combinators

[2] It is called combinatory logic for historical reasons; we shall not treat it as a logic.

satisfy a version of the Church-Rosser Theorem: if $P = Q$ (by any number of reductions, forwards or backwards) then there exists a term Z such that $P \twoheadrightarrow_w Z$ and $Q \twoheadrightarrow_w Z$.

7.3 Abstraction on Combinators

Any λ-term may be transformed into a roughly equivalent combinatory term. (The meaning of "roughly" is examined below.) The key is the transformation of a combinatory term P into another combinator term, written as $\lambda^* x.P$ since it behaves like a λ-abstraction.[3]

Definition 12 The operation $\lambda^* x$, where x is a variable, is defined recursively as follows:

$$\begin{array}{lcll} \lambda^* x.x & \equiv & \mathbf{I} & \\ \lambda^* x.P & \equiv & \mathbf{K}\,P & (x \text{ not free in } P) \\ \lambda^* x.P\,Q & \equiv & \mathbf{S}\,(\lambda^* x.P)(\lambda^* x.Q) & \end{array}$$

Finally, $\lambda^* x_1 \ldots x_n.P$ abbreviates $\lambda^* x_1.(\ldots \lambda^* x_n.P \ldots)$.

Note that $\lambda^* x$ is not part of the syntax of combinatory logic, but stands for the term constructed as according to the definition above. Here is an example of combinatory abstraction:

$$\begin{array}{rcl} \lambda^* x\,y.y\,x & \equiv & \lambda^* x.(\lambda^* y.y\,x) \\ & \equiv & \lambda^* x.\,\mathbf{S}\,(\lambda^* y.y)(\lambda^* y.x) \\ & \equiv & \lambda^* x.(\mathbf{S}\,\mathbf{I})(\mathbf{K}\,x) \\ & \equiv & \mathbf{S}\,(\lambda^* x.\,\mathbf{S}\,\mathbf{I})(\lambda^* x.\,\mathbf{K}\,x) \\ & \equiv & \mathbf{S}\,(\mathbf{K}\,(\mathbf{S}\,\mathbf{I}))(\mathbf{S}\,(\lambda^* x.\,\mathbf{K})(\lambda^* x.x)) \\ & \equiv & \mathbf{S}\,(\mathbf{K}\,(\mathbf{S}\,\mathbf{I}))(\mathbf{S}\,(\mathbf{K}\,\mathbf{K})\,\mathbf{I}) \end{array}$$

Each λ^* can *double* the number of applications in a term; in general, growth is exponential. Turner discovered a better abstraction method, discussed in the next section. First, let us show that combinatory abstraction behaves like its λ-calculus cousin. Let FV be defined for combinatory terms in an analogous manner to Definition 3.

Theorem 13 *For every combinatory term P we have*

$$\begin{array}{rcl} \mathrm{FV}(\lambda^* x.P) & = & \mathrm{FV}(P) - \{x\} \\ (\lambda^* x.P)x & \twoheadrightarrow_w & P \end{array}$$

[3] Some authors write $[x]P$ for $\lambda^* x.P$.

Proof We prove both properties independently, by structural induction on P. There are three cases.

If P is the variable x, then $\lambda^* x.x \equiv \mathbf{I}$. Clearly

$$\begin{aligned} &\mathrm{FV}(\lambda^* x.x) = \mathrm{FV}(\mathbf{I}) = \emptyset = \mathrm{FV}(x) - \{x\} \\ &(\lambda^* x.x)x \equiv \mathbf{I}\,x \twoheadrightarrow_w x \end{aligned}$$

If P is any term not containing x, then $\lambda^* x.P \equiv \mathbf{K}\,P$ and

$$\begin{aligned} &\mathrm{FV}(\lambda^* x.P) = \mathrm{FV}(\mathbf{K}\,P) = \mathrm{FV}(P) \\ &(\lambda^* x.P)x \equiv \mathbf{K}\,P x \to_w P \end{aligned}$$

If $P \equiv Q\,R$, and x is free in Q or R, then $\lambda^* x.P \equiv \mathbf{S}\,(\lambda^* x.Q)(\lambda^* x.R)$. This case is the inductive step and we may assume, as induction hypotheses, that the theorem holds for Q and R:

$$\begin{aligned} \mathrm{FV}(\lambda^* x.Q) &= \mathrm{FV}(Q) - \{x\} \\ \mathrm{FV}(\lambda^* x.R) &= \mathrm{FV}(R) - \{x\} \end{aligned}$$

$$\begin{aligned} (\lambda^* x.Q)x &\twoheadrightarrow_w Q \\ (\lambda^* x.R)x &\twoheadrightarrow_w R \end{aligned}$$

We now consider the set of free variables:

$$\begin{aligned} \mathrm{FV}(\lambda^* x.Q\,R) &= \mathrm{FV}(\mathbf{S}\,(\lambda^* x.Q)(\lambda^* x.R)) \\ &= (\mathrm{FV}(Q) - \{x\}) \bigcup (\mathrm{FV}(R) - \{x\}) \\ &= \mathrm{FV}(Q\,R) - \{x\} \end{aligned}$$

Finally, we consider application:

$$\begin{aligned} (\lambda^* x.P)x &\equiv \mathbf{S}\,(\lambda^* x.Q)(\lambda^* x.R)x \\ &\to_w (\lambda^* x.Q)x((\lambda^* x.R)x) \\ &\twoheadrightarrow_w Q((\lambda^* x.R)x) \\ &\twoheadrightarrow_w Q\,R \end{aligned}$$

□

Using $(\lambda^* x.P)x \twoheadrightarrow_w P$, we may derive an analogue of β-reduction for combinatory logic. We also get a strong analogue of α-conversion — changes in the abstraction variable are absolutely insignificant, yielding identical terms.

Theorem 14 *For all combinatory terms P and Q,*

$$\begin{aligned} (\lambda^* x.P)Q &\twoheadrightarrow_w P[Q/x] \\ \lambda^* x.P &\equiv \lambda^* y.P[y/x] \qquad \textit{if } y \notin \mathrm{FV}(P) \end{aligned}$$

Proof Both statements are routine structural inductions; the first can also be derived from the previous theorem by a general substitution theorem [1]. □

7.4 The Relation Between λ-Terms and Combinators

The mapping $(\ )_{CL}$ converts a λ-term into a combinator term. It simply applies λ^* recursively to all the abstractions in the λ-term; note that the innermost abstractions are performed first! The inverse mapping, $(\ )_\lambda$, converts a combinator term into a λ-term.

Definition 15 The mappings $(\ )_{CL}$ and $(\ )_\lambda$ are defined recursively as follows:

$$\begin{aligned}
(x)_{CL} &\equiv x \\
(M\ N)_{CL} &\equiv (M)_{CL}(N)_{CL} \\
(\lambda x.M)_{CL} &\equiv \lambda^* x.(M)_{CL}
\end{aligned}$$

$$\begin{aligned}
(x)_\lambda &\equiv x \\
(\mathbf{K})_\lambda &\equiv \lambda x\ y.x \\
(\mathbf{S})_\lambda &\equiv \lambda x\ y\ z.x\ z(y\ z) \\
(P\ Q)_\lambda &\equiv (P)_\lambda(Q)_\lambda
\end{aligned}$$

Different versions of combinatory abstraction yield different versions of $(\ )_{CL}$; the present one causes exponential blow-up in term size, but it is easy to reason about. Let us abbreviate $(M)_{CL}$ as M_{CL} and $(P)_\lambda$ as P_λ. It is easy to check that $(\ )_{CL}$ and $(\ )_\lambda$ do not add or delete free variables:

$$\mathrm{FV}(M) = \mathrm{FV}(M_{CL}) \qquad \mathrm{FV}(P) = \mathrm{FV}(P_\lambda)$$

Equality is far more problematical. The mappings do give a tidy correspondence between the λ-calculus and combinatory logic, provided we assume the principle of *extensionality*. This asserts that two functions are equal if they return equal results for every argument value. In combinatory logic, extensionality takes the form of a new rule for proving equality:

$$\frac{Px = Qx}{P = Q} \qquad (x \text{ not free in } P \text{ or } Q)$$

In the λ-calculus, extensionality can be expressed by a similar rule or by introducing η-reduction:

$$\lambda x.Mx \to_\eta M \qquad (x \text{ not free in } M)$$

Assuming extensionality, the mappings preserve equality [1]:

$$\begin{array}{lcll}
(M_{CL})_\lambda & = & M & \text{in the } \lambda\text{-calculus} \\
(P_\lambda)_{CL} & = & P & \text{in combinatory logic} \\
M = N & \iff & M_{CL} = N_{CL} & \\
P = Q & \iff & P_\lambda = Q_\lambda &
\end{array}$$

Normal forms and reductions are not preserved. For instance, $\mathbf{S}\ \mathbf{K}$ is a normal form of combinatory logic; no weak reductions apply to it. But the corresponding λ-term is not in normal form:

$$(\mathbf{S}\ \mathbf{K})_\lambda \equiv (\lambda x\ y\ z.x\ z(y\ z))(\lambda x\ y.x) \twoheadrightarrow \lambda y\ z.z$$

There are even combinatory terms in normal form whose corresponding λ-term has no normal form! Even where both terms follow similar reduction sequences, reductions in combinatory logic have much finer granularity than those in the λ-calculus; consider how many steps are required to simulate a β-reduction in combinatory logic.

Normal forms are the outputs of functional programs; surely, they ought to be preserved. Reduction is the process of generating the outputs. Normally we should not worry about this, but lazy evaluation has to deal with infinite outputs that cannot be fully evaluated. Thus, the rate and granularity of reduction is important. Despite the imperfect correspondence between λ-terms and combinators, compilers based upon combinatory logic appear to work. Perhaps the things not preserved are insignificant for computational purposes. More research needs to be done in the operational behaviour of functional programs.

8 Compiling Methods Using Combinators

Combinator abstraction gives us a theoretical basis for removing variables from λ-terms, and will allow efficient graph reduction. But first, we require a mapping from λ-terms to combinators that generates more compact results. Recall that λ^* causes exponential blowup:

$$\lambda^* x\ y.y\ x \equiv \mathbf{S}\,(\mathbf{K}\,(\mathbf{S}\ \mathbf{I}))(\mathbf{S}\,(\mathbf{K}\ \mathbf{K})\,\mathbf{I})$$

The improved version of combinatory abstraction relies on two new combinators, $\mathbf{B}$ and $\mathbf{C}$, to handle special cases of $\mathbf{S}$:

$$\begin{array}{lll} \mathbf{B}\ P\ Q\ R & \rightarrow_w & P(Q\ R) \\ \mathbf{C}\ P\ Q\ R & \rightarrow_w & P\ R\ Q \end{array}$$

Note that $\mathbf{B}\ P\ Q\ R$ yields the function composition of P and Q. Let us call the new abstraction mapping λ^T, after David Turner, its inventor:

$$\begin{array}{lcll} \lambda^T x.x & \equiv & \mathbf{I} & \\ \lambda^T x.P & \equiv & \mathbf{K}\ P & (x \text{ not free in } P) \\ \lambda^T x.P\ x & \equiv & P & (x \text{ not free in } P) \\ \lambda^T x.P\ Q & \equiv & \mathbf{B}\ P(\lambda^T x.Q) & (x \text{ not free in } P) \\ \lambda^T x.P\ Q & \equiv & \mathbf{C}\ (\lambda^T x.P)Q & (x \text{ not free in } Q) \\ \lambda^T x.P\ Q & \equiv & \mathbf{S}\ (\lambda^T x.P)(\lambda^T x.Q) & (x \text{ free in } P \text{ and } Q) \end{array}$$

Although λ^T is a bit more complicated than λ^*, it generates much better code (i.e. combinators). The third case, for $P\,x$, takes advantage of extensionality; note its similarity to η-reduction. The next two cases abstract over $P\,Q$ according to whether or not the abstraction variable is actually free in P or Q. Let us do our example again:

$$\begin{aligned}
\lambda^T x\,y.y\,x &\equiv \lambda^T x.(\lambda^T y.y\,x) \\
&\equiv \lambda^T x.\,\mathbf{C}\,(\lambda^T y.y)x \\
&\equiv \lambda^T x.\,\mathbf{C}\;\mathbf{I}\,x \\
&\equiv \mathbf{C}\;\mathbf{I}
\end{aligned}$$

The size of the generated code has decreased by a factor of four! Here is another example, from Paper 6 of the 1993 Examination. Let us translate the λ-encoding of the ordered pair operator:

$$\begin{aligned}
\lambda^T x.\lambda^T y.\lambda^T f.f\,x\,y &\equiv \lambda^T x.\lambda^T y.\,\mathbf{C}\,(\lambda^T f.f\,x)\,y \\
&\equiv \lambda^T x.\lambda^T y.\,\mathbf{C}\,(\mathbf{C}\,(\lambda^T f.f)\,x)\,y \\
&\equiv \lambda^T x.\lambda^T y.\,\mathbf{C}\,(\mathbf{C}\;\mathbf{I}\;x)\,y \\
&\equiv \lambda^T x.\,\mathbf{C}\,(\mathbf{C}\;\mathbf{I}\;x) \\
&\equiv \mathbf{B}\;\mathbf{C}\,(\lambda^T x.\,\mathbf{C}\;\mathbf{I}\;x) \\
&\equiv \mathbf{B}\;\mathbf{C}\,(\mathbf{C}\;\mathbf{I}).
\end{aligned}$$

Unfortunately, λ^T can still cause a quadratic blowup in code size; additional primitive combinators should be introduced (See Field and Harrison [4, page 286]. Furthermore, all the constants of the functional language — numbers, arithmetic operators, ... — must be taken as primitive combinators.

Introducing more and more primitive combinators makes the code smaller and faster. This leads to the method of *super combinators*, where the set of primitive combinators is extracted from the program itself.

Exercise 12 Show $\mathbf{B}\,P\,\mathbf{I} = P$ using extensionality.

Exercise 13 Verify that $\mathbf{C}\;\mathbf{I}$ behaves like the λ-term $\lambda x\,y.y\,x$ when applied to two arguments.

Exercise 14 What would $\lambda^T x\,y.y\,x$ yield if we did not apply the third case in the definition of λ^T?

8.1 Combinator Terms as Graphs

Consider the ISWIM program

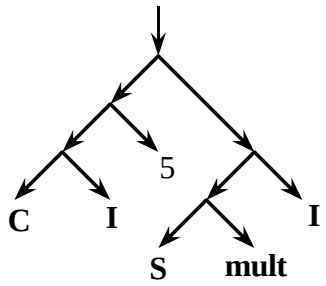
$$\mathit{let}\ sqr(n) = n \times n\ \mathbf{in}\ sqr(5)$$

Let us translate it to combinators:

$$\begin{aligned}(\lambda^T f.\, f\,5)(\lambda^T n.\,\mathbf{mult}\,n\,n) &\equiv \mathbf{C}\ \mathbf{I}\,5\,(\mathbf{S}\,(\lambda^T n.\,\mathbf{mult}\,n)(\lambda^T n.n)) \\ &\equiv \mathbf{C}\ \mathbf{I}\,5\,(\mathbf{S}\ \mathbf{mult}\ \mathbf{I})\end{aligned}$$

This is a *closed* term — it contains no free variables (and no bound variables, of course). Therefore it can be evaluated by reducing it to normal form.

Graph reduction works on the combinator term's graph structure. This resembles a binary tree with branching at each application. The graph structure for **C I** 5 (**S mult I**) is as follows:



Repeated arguments cause sharing in the graph, ensuring that they are never evaluated more than once.

8.2 The Primitive Graph Transformations

Graph reduction deals with terms that contain no variables. Each term, and its subterms, denote constant values. Therefore we may transform the graphs destructively — operands are never copied. The graph is *replaced* by its normal form!

The primitive combinators reduce as shown in Figure 1. The sharing in the reduction for **S** is crucial, for it avoids copying R.

We also require graph reduction rules for the built-in functions, such as **mult**. Because **mult** is a strict function, the graph for **mult** P Q can only be reduced after P and Q have been reduced to numeric constants m and n. Then **mult** m n is replaced by the constant whose value is $m \times n$. Graph reduction proceeds by walking down the graph's leftmost branch, seeking something to reduce. If the leftmost symbol is a combinator like **I**, **K**, **S**, **B**, or **C**, with the requisite number of operands, then it applies the corresponding transformation. If the leftmost

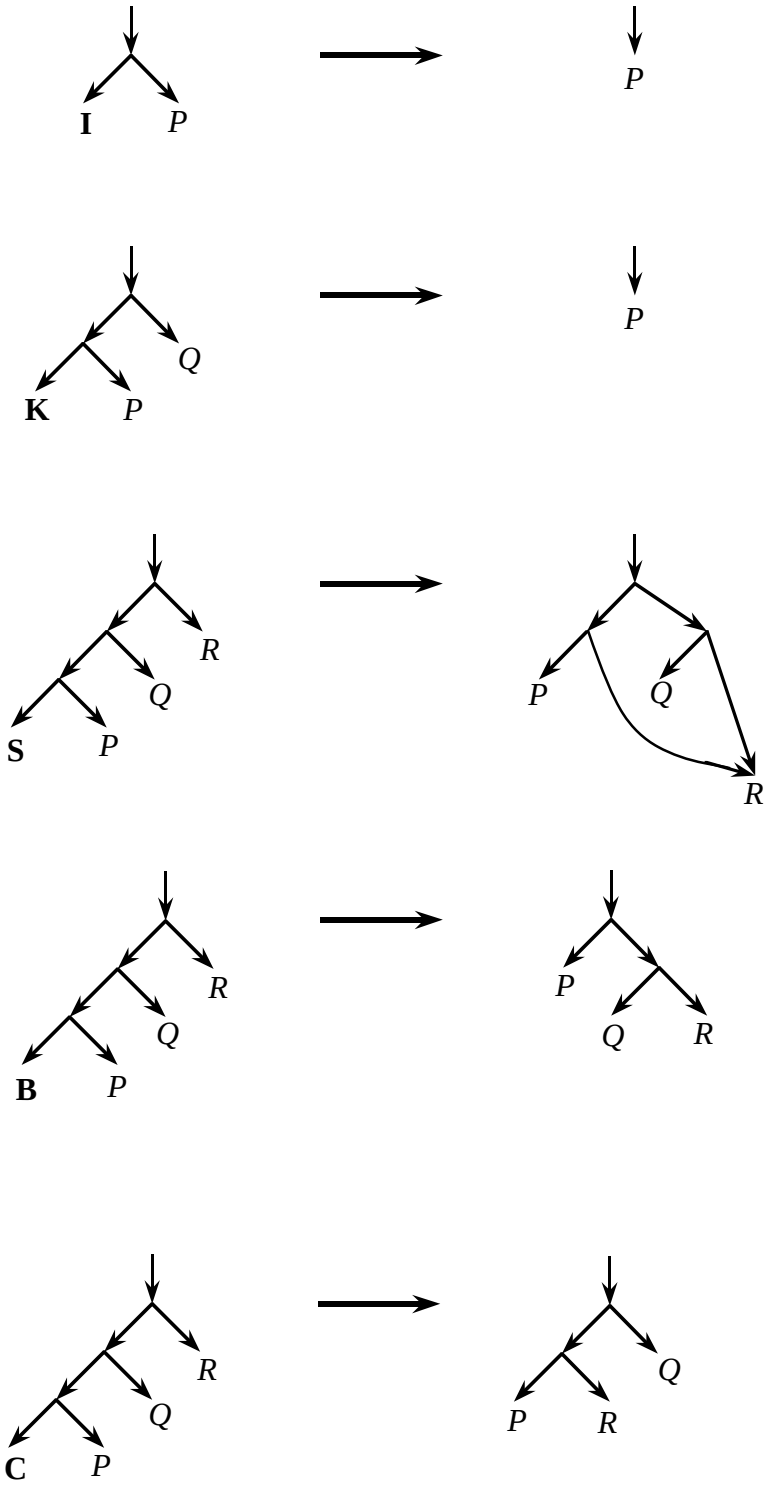


Figure 1: Graph reduction for combinators

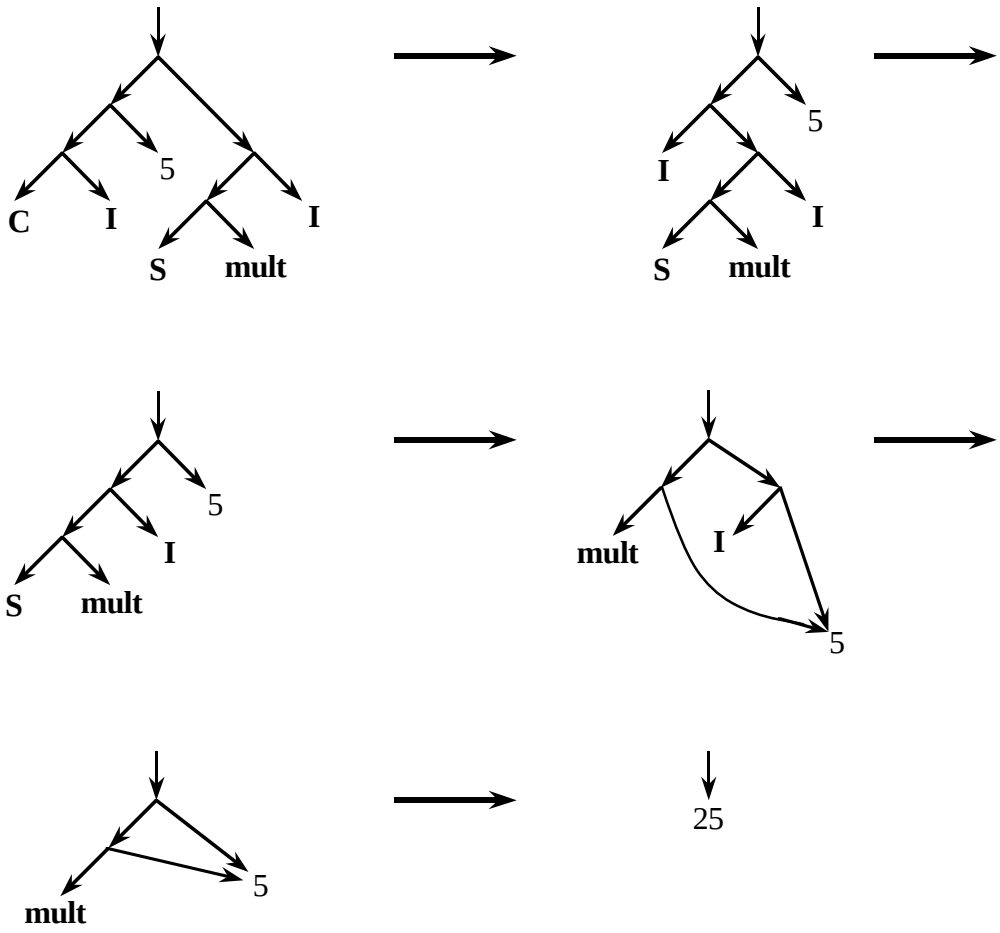


Figure 2: A graph reduction sequence

symbol is a strict combinator like **mult**, then it recursively traverses the operands, attempting to reduce them to numbers.

Figure 2 presents the graph reduction sequence for the ISWIM program

$$let\ sqr(n) = n \times n\ \textbf{in}\ sqr(5).$$

The corresponding term reductions are as follows:

$$\begin{aligned}
\mathbf{C}\ \mathbf{I}\,5\,(\mathbf{S}\ \mathbf{mult}\ \mathbf{I}) &\rightarrow \mathbf{I}\ (\mathbf{S}\ \mathbf{mult}\ \mathbf{I})\,5\\
&\rightarrow \mathbf{S}\ \mathbf{mult}\ \mathbf{I}\,5\\
&\rightarrow \mathbf{mult}\,5\,(\mathbf{I}\,5)\\
&\rightarrow \mathbf{mult}\,5\,5\\
&\rightarrow 25
\end{aligned}$$

Clearly visible in the graphs, but not in the terms, is that the two copies of 5 are shared. If, instead of 5, the argument of *sqr* had been a large combinatory term P compiled from the program, then P would have been evaluated only once. Graph reduction can also discard terms by the rule $\mathbf{K}\,P\,Q \rightarrow_w P$; here Q is never evaluated.

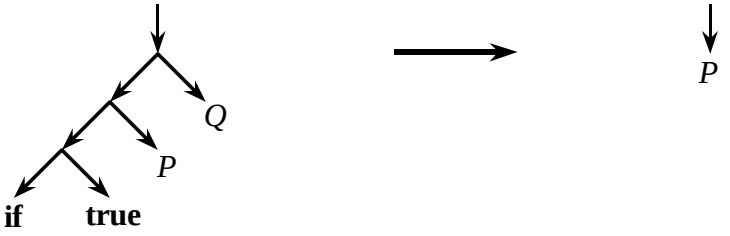
8.3 Booleans and Pairing

The λ-calculus encodings of ordered pairs, Church numerals and so forth work with combinators, but are impractical to use for compiling a functional language. New combinators and new reductions are introduced instead.

With lazy evaluation, if-then-else can be treated like a function, with the two reductions

$$\begin{aligned}
\mathbf{if}\ \mathbf{true}\ P\ Q &\rightarrow_w P\\
\mathbf{if}\ \mathbf{false}\ P\ Q &\rightarrow_w Q.
\end{aligned}$$

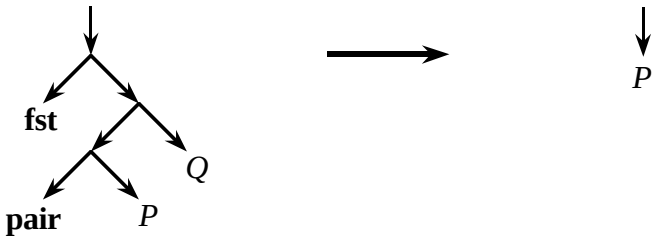
These reductions discard P or Q if it is not required; there is no need for tricks to delay their evaluation. The first reduction operates on graphs as shown.



Pairing is also lazy, as it is in the λ-calculus; we introduce the reductions

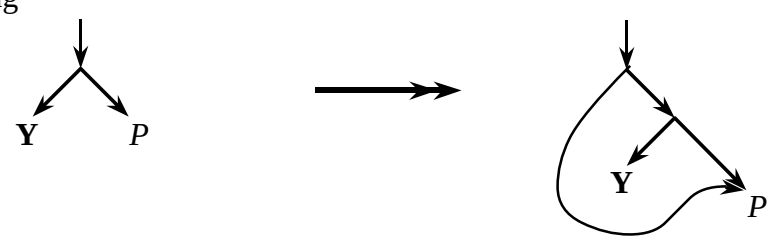
$$\begin{aligned}
\mathbf{fst}\,(\mathbf{pair}\ P\ Q) &\rightarrow_w P\\
\mathbf{snd}\,(\mathbf{pair}\ P\ Q) &\rightarrow_w Q.
\end{aligned}$$

The corresponding graph reductions should be obvious:

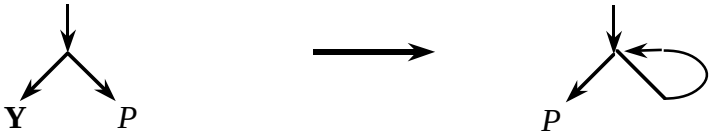


8.4 Recursion: Cyclic Graphs

Translating **Y** into combinator form will work, yielding a mult-step reduction resembling[4]



This is grossly inefficient; **Y** must repeat its work at every recursive invocation! Instead, take **Y** as a primitive combinator satisfying $\mathbf{Y}\,P \to_w P(\mathbf{Y}\,P)$ and adopt a graph reduction rule that replaces the **Y** by a *cycle*:



Since P is never copied, reductions that occur in it yield permanent simplifications — they are not repeated when the function is entered recursively.

To illustrate this, consider the ISWIM program

$$\mathit{letrec}\ \mathit{from}(n) = \mathbf{pair}\,n(\mathit{from}(1+n))\ \mathbf{in}\ \mathit{from}(1).$$

The result should be the infinite list $(1, (2, (3, \ldots)))$. We translate *from* into combinators, starting with

$$\mathbf{Y}\,(\lambda^T\, f\ n.\,\mathbf{pair}\,n(\,f\,(\,\mathbf{add}\,1\,n))$$

and obtain (verify this)

$$\mathbf{Y}\,(\,\mathbf{B}\,(\,\mathbf{S}\ \mathbf{pair}\,)(\,\mathbf{C}\ \mathbf{B}\,(\,\mathbf{add}\,1)))$$

Figures 3 and 4 give the graph reductions. A cyclic node, labelled θ, quickly appears. Its rather tortuous transformations generate a recursive occurrence of

[4]The picture is an over-simplification; recall that we do not have $\mathbf{Y}\,P \twoheadrightarrow P(\mathbf{Y}\,P)$!

from deeper in the graph. The series of reductions presumes that the environment is demanding evaluation of the result; in lazy evaluation, nothing happens until it is forced to happen.

Graph reduction will leave the term **add** 1 1 unevaluated until something demands its value; the result of *from*(1) is really $(1, (1+1, (1+1+1, \ldots)))$. Graph reduction works a bit like macro expansion. Non-recursive function calls get expanded once and for all the first time they are encountered; thus, programmers are free to define lots of simple functions in order to aid readability. Similarly, constant expressions are evaluated once and for all when they are first encountered. Although this behaviour avoids wasteful recomputation, it can cause the graph to grow and grow, consuming all the store — a *space leak*. The displayed graph reduction illustrates how this could happen.

Exercise 15 Translate **Y** to combinators and do some steps of the reduction of **Y** P.

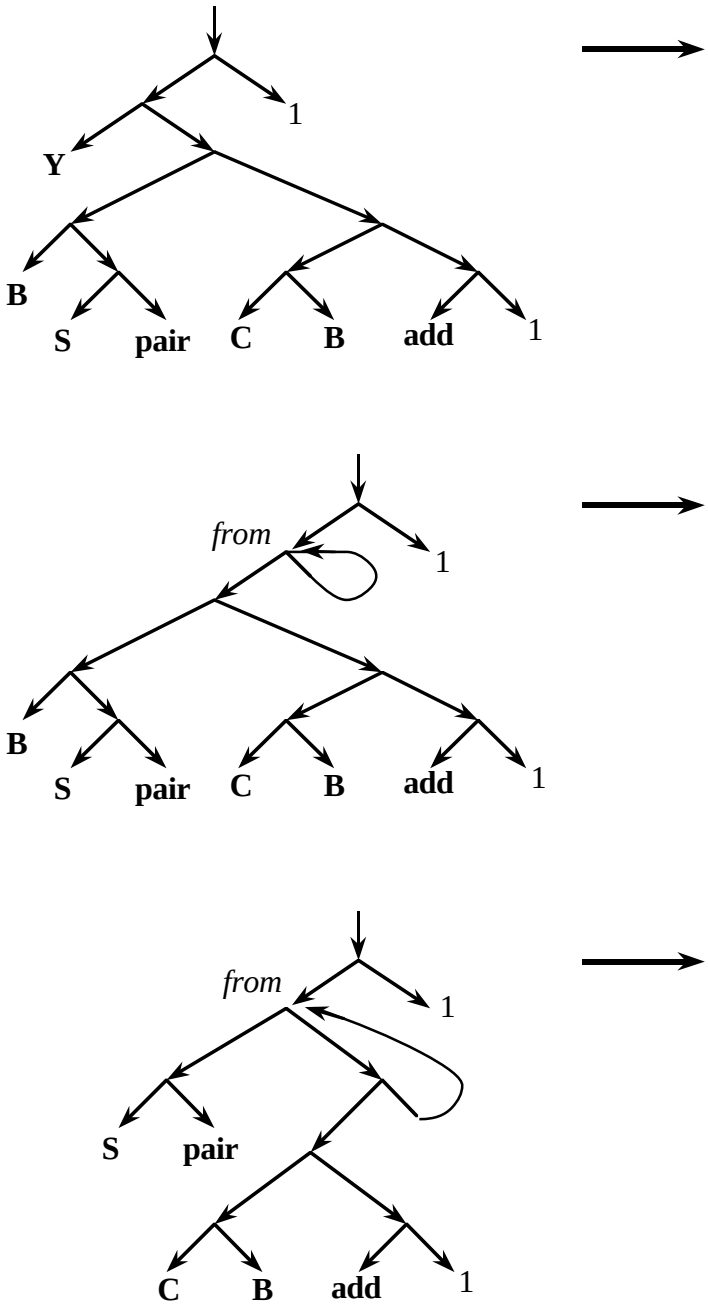


Figure 3: Reductions involving recursion

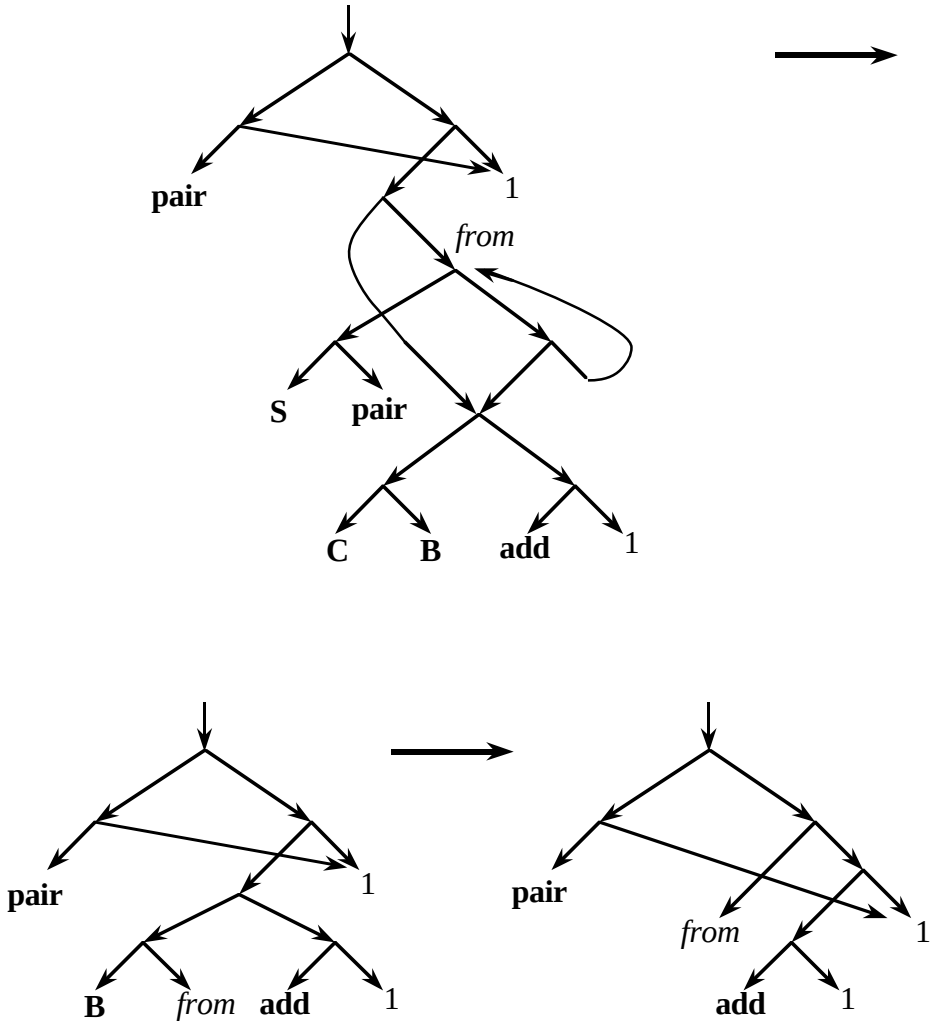


Figure 4: Reductions involving recursion (continued)

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[3] G. Cousineau and G. Huet. The CAML primer. Technical report, INRIA, Rocquencourt, France, 1990.

[4] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[5] Michael J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall, 1988.

[6] J. Roger Hindley and Jonathon P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[7] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, January 1964.

[8] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[9] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

Demonstrating Lambda Calculus Reduction

Peter Sestoft

Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark

and

IT University of Copenhagen, Denmark

sestoft@dina.kvl.dk

Abstract. We describe lambda calculus reduction strategies, such as call-by-value, call-by-name, normal order, and applicative order, using big-step operational semantics. We show how to simply and efficiently trace such reductions, and use this in a web-based lambda calculus reducer available at ⟨http://www.dina.kvl.dk/~sestoft/lamreduce/⟩.

1 Introduction

The pure untyped lambda calculus is often taught as part of the computer science curriculum. It may be taught in a computability course as a classical computation model. It may be taught in a semantics course as the foundation for denotational semantics. It may be taught in a functional programming course as the archetypical minimal functional programming language. It may be taught in a programming language course for the same reason, or to demonstrate that a very small language can be universal, e.g. can encode arithmetics (as well as data structures, recursive function definitions and so on), using encodings such as these:

$$\begin{aligned} \mathit{two} &\equiv \lambda f.\lambda x.f(fx) \\ \mathit{four} &\equiv \lambda f.\lambda x.f(f(f(fx))) \\ \mathit{add} &\equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx) \end{aligned} \tag{1}$$

This paper is motivated by the assumption that to appreciate the operational aspects of pure untyped lambda calculus, students must experiment with it, and that tools encourage experimentation with encodings and reduction strategies by making it less tedious and more fun.

In this paper we describe a simple way to create a tool for demonstrating lambda calculus reduction. Instead of describing a reduction strategy by a procedure for locating the next redex to be contracted, we describe it by a big-step operational semantics. We show how to trace the β-reductions performed during reduction.

To do this we also precisely define and clarify the relation between programming language concepts such as call-by-name and call-by-value, and lambda calculus concepts such as normal order reduction and applicative order reduction. These have been given a number of different interpretations in the literature.

In T. Mogensen, D. Schmidt, I. H. Sudburough (editors): *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones.* Lecture Notes in Computer Science 2566, pages 420-435. Springer-Verlag 2002.

2 Motivation and Related Work

Much has been written about the lambda calculus since Church developed it as a foundation for mathematics [6]. Landin defined the semantics of programming languages in terms of the lambda calculus [11], and gave a call-by-value interpreter for it: the SECD-machine [10]. Strachey used lambda calculus as a meta-language for denotational semantics, and Scott gave models for the pure untyped lambda calculus, making sure that self-application could be assigned a meaning; see Stoy [22]. Self-application $(x\,x)$ of a term x is used when encoding recursion, for instance in Church's Y combinator:

$$Y \equiv \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \tag{2}$$

Plotkin studied the call-by-value lambda calculus corresponding to the functional language ISWIM [12] implemented by Landin's SECD-machine, and also a related call-by-name lambda calculus, and observed that one characteristic of a functional programming language was the absence of reduction under lambda abstractions [19].

Barendregt [4] is the standard reference on the untyped lambda calculus, with emphasis on models and proof theory, not programming languages.

Many textbooks on functional programming or denotational semantics present the pure untyped lambda calculus, show how to encode numbers and algebraic data types, and define evaluators for it. One example is Paulson's ML textbook [16], which gives interpreters for call-by-name as well as call-by-value.

So is there really a need for yet another paper on lambda calculus reduction? We do think so, because it is customary to look at the lambda calculus either from the programming language side or from the calculus or model side, leaving the relations between the sides somewhat unclear.

For example, Plotkin [19] defines call-by-value reduction as well as call-by-name reduction, but the call-by-name rules take free variables into account only to a limited extent. By the rules, $x\,((\lambda z.z)\,v)$ reduces to $x\,v$, but $(x\,y)\,((\lambda z.z)\,v)$ does not reduce to $x\,y\,v$ [19, page 146]. Similarly, the call-by-value strategy described by Felleisen and Hieb using evaluation contexts [8, Section 2] would not reduce $(x\,y)\,((\lambda z.z)\,v)$ to $x\,y\,v$, since there is no evaluation context of the form $(x\,y)\,[\,]$. This is unproblematic because, following Landin, these researchers were interested only in terms with no free variables, and in reduction only outside lambda abstractions.

But it means that the reduction rules are not immediately useful for terms that have free variables, and therefore not useful for experimentation with the terms that result from encoding programming language constructs in the pure lambda calculus.

Conversely, Paulson [16] presents call-by-value and call-by-name interpreters for the pure lambda calculus that do handle free variables. However, they also perform reduction under lambda abstractions (unlike functional programming languages), and the evaluation order is not leftmost outermost: under call-by-name, an application $(e_1\,e_2)$ is reduced by first reducing e_1 to head normal form,

so redexes inside e_1 may be contracted before an enclosing leftmost redex. This makes the relation between Paulson's call-by-name and normal order (leftmost outermost) reduction strategies somewhat unclear.

Therefore we find that it may be useful to contrast the various reduction strategies, present them using big-step operational semantics, present their (naive) implementation in ML, and show how to obtain a trace of the reduction.

3 The Pure Untyped Lambda Calculus

We use the pure untyped lambda calculus [4]. A lambda term is a *variable* x, a lambda *abstraction* $\lambda x.e$ which binds x in e, or an *application* $(e_1\, e_2)$ of a 'function' e_1 to an 'argument' e_2:

$$e ::= \; x \mid \lambda x.e \mid e_1\, e_2 \tag{3}$$

Application associates to the left, so $(e_1\, e_2\, e_3)$ means $((e_1\, e_2)\, e_3)$. A lambda term may have free variables, not bound by any enclosing lambda abstraction. Term identity $e_1 \equiv e_2$ is taken modulo renaming of lambda-bound variables. The notation $e[e_x/x]$ denotes substitution of e_x for x in e, with renaming of bound variables in e if necessary to avoid capture of free variables in e_x.

A *redex* is a subterm of the form $((\lambda x.e)\, e_2)$; the *contraction* of a redex produces $e[e_2/x]$, substituting the argument e_2 for every occurrence of the parameter x in e. By $e \longrightarrow_\beta e'$ we denote *β-reduction*, the contraction of some redex in e to obtain e'.

A redex is to the *left* of another redex if its lambda abstractor appears further to the left. The *leftmost outermost* redex (if any) is the leftmost redex not contained in any other redex. The *leftmost innermost* redex (if any) is the leftmost redex not containing any other redex.

4 Functional Programming Languages

In practical functional programming languages such as Scheme [20], Standard ML [14] or Haskell [18], programs cannot have free variables, and reductions are not performed under lambda abstractions or other variable binders, because this would considerably complicate their efficient implementation [17].

However, an implementation of lambda calculus reduction must perform reductions under lambda abstractions. Otherwise, *add two two* would not reduce to *four* using the encodings (1), which would disappoint students.

Because free variables and reduction under abstraction are absent in functional languages, it is unclear what the programming language concepts call-by-value and call-by-name mean in the lambda calculus. In particular, how should free variables be handled, and to what normal form should call-by-value and call-by-name evaluate? We propose the following answers:

– A free variable is similar to a data constructor (in Standard ML or Haskell), that is, an uninterpreted function symbol. If the free variable x is in function position $(x\, e_2)$, then call-by-value should reduce the argument expression e_2, whereas call-by-name should not. This is consistent with constructors being strict in strict languages (e.g. ML) and non-strict in non-strict languages (e.g. Haskell).
– Functional languages perform no reduction under abstractions, and thus reduce terms to weak normal forms only. In particular, call-by-value reduces to weak normal form, and call-by-name reduces to weak head normal form. Section 6 define these normal forms.

5 Lazy Functional Programming Languages

Under lazy evaluation, a variable-bound term is evaluated at most once, regardless how often the variable is used [17]. Thus an argument term may not be duplicated before it has been reduced, and may be reduced only if actually used. This evaluation mechanism may be called call-by-need, or call-by-name with sharing of argument evaluation. The equational theory of call-by-need lambda calculus has been studied by Ariola and Felleisen [2] among others. (By contrast, the lazy lambda calculus of Abramsky and Ong [1] is not lazy in the sense discussed here; rather, it is the theory of call-by-name lambda calculus, without reduction under abstractions.)

Lazy functional languages also permit the creation of cyclic terms, or cycles in the heap. For instance, this declaration creates a finite (cyclic) representation of an infinite list of 1's:

```
val ones = 1 :: ones
```

Thus to be true also to the intensional properties of lazy languages (such as time and space consumption), a model should be able to describe such constant-size cyclic structures. Substitution of terms for variables cannot truly model them, only approximate them by unfolding of a recursive term definition, possibly encoded using a recursion combinator such as (2). To properly express sharing of subterm evaluation, and the creation of cyclic terms, one must extend the syntax (3) with mutually recursive bindings:

$$e ::= \; x \mid \lambda x.e \mid e\, e \mid \mathit{letrec}\ \{x_i = e_i\}\ \mathit{in}\ e \tag{4}$$

The sharing of subterm evaluation and the dynamic creation of cyclic terms may be modelled using graph reduction, as suggested by Wadsworth [24] and used in subsequent work [3, 17, 23], or using an explicit heap [13, 21].

Thus a proper modelling of lazy evaluation, with sharing of argument evaluation and cyclic data structures, requires syntactic extensions as well as a more elaborate evaluation model than just term rewriting. We shall not consider lazy evaluation any further in this paper, and shall consider only the syntax in (3) above.

6 Normal Forms

We need to distinguish four different normal forms, depending on whether we reduce under abstractions or not (in functional programming languages), and depending on whether we reduce the arguments before substitution (in strict languages) or not (in non-strict languages).

Figure 1 summarizes the four normal forms using four context-free grammars. In each grammar, the symbol E denotes a term in the relevant normal form, e denotes an arbitrary lambda term generated by (3), and $n \geq 0$. Note how the two dichotomies generate the four normal forms just by varying the form of lambda abstraction bodies and application arguments.

	Reduce under abstractions	
Reduce args	**Yes**	**No**
Yes	Normal form $E ::= \lambda x.E \mid x\,E_1 \ldots E_n$	Weak normal form $E ::= \lambda x.e \mid x\,E_1 \ldots E_n$
No	Head normal form $E ::= \lambda x.E \mid x\,e_1 \ldots e_n$	Weak head normal form $E ::= \lambda x.e \mid x\,e_1 \ldots e_n$

Fig. 1. Normal forms. The e_i denote arbitrary lambda terms generated by (3).

7 Reduction Strategies and Reduction Functions

We present several reduction strategies using big-step operational semantics, or natural semantics [9], and their implementation in Standard ML. The premises of each semantic rule are assumed to be evaluated from left to right, although this is immaterial to their logical interpretation. We exploit that Standard ML evaluates a function's arguments before calling the function, evaluates the right-hand side of `let`-bindings before binding the variable, and evaluates subterms from left to right [14].

We model lambda terms x, $\lambda x.e$ and $(e\,e)$ as ML constructed data, representing variable names by strings:

```
datatype lam = Var of string
             | Lam of string * lam
             | App of lam * lam
```

We also assume an auxiliary function `subst : lam -> lam -> lam` that implements capture-free substitution, so `subst ex (Lam(x, e))` is the ML representation of $e[e_x/x]$, the result of contracting the redex $(\lambda x.e)\,e_x$.

7.1 Call-by-Name Reduction to Weak Head Normal Form

Call-by-name reduction $e \xrightarrow{bn} e'$ reduces the leftmost outermost redex not inside a lambda abstraction first. It treats free variables as non-strict data constructors. For terms without free variables, it coincides with Plotkin's call-by-name reduction [19, Section 5], and is closely related to Engelfriet and Schmidt's outside-in derivation (in context-free tree grammars, or first-order recursion equations) [7, page 334].

$$x \xrightarrow{bn} x$$

$$(\lambda x.e) \xrightarrow{bn} (\lambda x.e)$$

$$\frac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{bn} e'}{(e_1\ e_2) \xrightarrow{bn} e'} \tag{5}$$

$$\frac{e_1 \xrightarrow{bn} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{bn} (e_1'\ e_2)}$$

It is easy to see that all four rules generate terms in weak head normal form. In particular, in the last rule e_1' must have form $y\, e_{11}' \ldots e_{1n}'$ for some $n \geq 0$, so $(e_1'\, e_2)$ is a weak head normal form. Assuming that the rule premises are read and 'executed' from left to right, it is also clear that only leftmost redexes are contracted. No reduction is performed under abstractions.

The following ML function `cbn` computes the weak head normal form of a lambda term, contracting redexes in the order implied by the operational semantics (5) above:

```
fun cbn (Var x)       = Var x
  | cbn (Lam(x, e))   = Lam(x, e)
  | cbn (App(e1, e2)) =
    case cbn e1 of
        Lam (x, e) => cbn (subst e2 (Lam(x, e)))
      | e1'        => App(e1', e2)
```

The first function clause above handles variables x and implements the first semantics rule. Similarly, the second function clause handles lambda abstractions $(\lambda x.e)$ and implements the second semantics rule. In both cases, the given term is returned unmodified. The third function clause handles applications $(e_1\, e_2)$ and implements the third and fourth semantics rule by discriminating on the result of reducing e_1. If the result is a lambda abstraction $(\lambda x.e)$ then the `cbn` function is called to reduce the expression $e[e_2/x]$; but if the result is any other expression e_1', the application $(e_1'\, e_2)$ is returned.

In all cases, this is exactly what the semantics rules in (5) describe. In fact, one can see that $e \xrightarrow{bn} e'$ if and only if `cbn` e terminates and returns e'.

7.2 Normal Order Reduction to Normal Form

Normal order reduction $e \xrightarrow{no} e'$ reduces the leftmost outermost redex first. In an application $(e_1\, e_2)$ the function term e_1 must be reduced using call-by-name (5). Namely, if e_1 reduces to an abstraction $(\lambda x.e)$, then the redex $((\lambda x.e)\, e_2)$ must be reduced before redexes in e, if any, because they would not be outermost.

$$
\begin{gathered}
x \xrightarrow{no} x \\
\frac{e \xrightarrow{no} e'}{(\lambda x.e) \xrightarrow{no} (\lambda x.e')} \\
\frac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{no} e'}{(e_1\ e_2) \xrightarrow{no} e'} \\
\frac{e_1 \xrightarrow{bn} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{no} e_1'' \qquad e_2 \xrightarrow{no} e_2'}{(e_1\ e_2) \xrightarrow{no} (e_1''\ e_2')}
\end{gathered}
\tag{6}
$$

It is easy to see that these rules generate normal form terms only. In particular, in the last rule e_1' must have form $y\, e_{11}' \ldots e_{1n}'$ for some $n \geq 0$, so e_1'' must have form $y\, E_{11}'' \ldots E_{1n}''$ for some normal forms E_{1i}'', and therefore $(e_1''\, e_2')$ is a normal form. Any redex contracted is the leftmost one not contained in any other redex; this relies on the use of call-by-name in the application rules. Reductions are performed also under lambda abstractions. Normal order reduction is *normalizing*: if the term e has a normal form, then normal order reduction of e will terminate (with the normal form as result).

The Standard ML function `nor : lam -> lam` below implements the reduction strategy. Note that it uses the function `cbn` defined in Section 7.1:

```
fun nor (Var x)       = Var x
  | nor (Lam (x, e))  = Lam(x, nor e)
  | nor (App(e1, e2)) =
    case cbn e1 of
        Lam(x, e) => nor (subst e2 (Lam(x, e)))
      | e1'       => let val e1'' = nor e1'
                     in App(e1'', nor e2) end
```

Again the first two cases of the function implement the first two reduction rules. The third case implements the third and fourth rules by evaluating e_1 using call-by-name `cbn` and then discriminating on whether the result is a lambda abstraction or not, as in the third and fourth rule in (6).

7.3 Call-by-Value Reduction to Weak Normal Form

Call-by-value reduction $e \xrightarrow{bv} e'$ reduces the leftmost innermost redex not inside a lambda abstraction first. It treats free variables as strict data constructors. For terms without free variables, it coincides with call-by-value reduction as defined by Plotkin [19, Section 4] and Felleisen and Hieb [8]. It is closely related to Engelfriet and Schmidt's inside-out derivations (in context-free tree grammars, or first-order recursion equations) [7, page 334]. It differs from call-by-name (Section 7.1) only by reducing the argument e_2 of an application $(e_1\, e_2)$ before contracting the redex, and before building an application term:

$$\begin{gathered} x \xrightarrow{bv} x \\ (\lambda x.e) \xrightarrow{bv} (\lambda x.e) \\ \frac{e_1 \xrightarrow{bv} (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2' \qquad e[e_2'/x] \xrightarrow{bv} e'}{(e_1\ e_2) \xrightarrow{bv} e'} \\ \frac{e_1 \xrightarrow{bv} e_1' \not\equiv (\lambda x.e) \qquad e_2 \xrightarrow{bv} e_2'}{(e_1\ e_2) \xrightarrow{bv} (e_1'\ e_2')} \end{gathered} \tag{7}$$

It is easy to see that these rules generate weak normal form terms only. In particular, in the last rule e_1' must have form $y\, E_{11}' \ldots E_{1n}'$ for some $n \geq 0$ and weak normal forms E_{1i}', and therefore $(e_1'\, e_2')$ is a weak normal form too. No reductions are performed under lambda abstractions. This is Paulson's `eval` auxiliary function [16, page 390]. The implementation of the rules by an ML function is straightforward and is omitted.

7.4 Applicative Order Reduction to Normal Form

Applicative order reduction $e \xrightarrow{ao} e'$ reduces the leftmost innermost redex first. It differs from call-by-value (Section 7.3) only by reducing also under abstractions:

$$\begin{gathered} x \xrightarrow{ao} x \\ \frac{e \xrightarrow{ao} e'}{(\lambda x.e) \xrightarrow{ao} (\lambda x.e')} \\ \frac{e_1 \xrightarrow{ao} (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2' \qquad e[e_2'/x] \xrightarrow{ao} e'}{(e_1\ e_2) \xrightarrow{ao} e'} \\ \frac{e_1 \xrightarrow{ao} e_1' \not\equiv (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2'}{(e_1\ e_2) \xrightarrow{ao} (e_1'\ e_2')} \end{gathered} \tag{8}$$

It is easy to see that the rules generate only normal form terms. As before, note that in the last rule e'_1 must have form $y\, E'_{11} \ldots E'_{1n}$ for some $n \geq 0$ and normal forms E'_{1i}. Also, it is clear that when a redex $((\lambda x.e)\, e'_2)$ is contracted, it contains no other redex, and it is the leftmost redex with this property.

Applicative order reduction is not normalizing; with $\Omega \equiv (\lambda x.(x\, x))(\lambda x.(x\, x))$ it produces an infinite reduction $((\lambda x.y)\, \Omega) \longrightarrow_\beta ((\lambda x.y)\, \Omega) \longrightarrow_\beta \ldots$ although the term has normal form y.

In fact, applicative order reduction fails to normalize applications of functions defined using recursion combinators, even with recursion combinators designed for call-by-value, such as Y_v:

$$Y_v \equiv \lambda h.(\lambda x.\lambda a.h\,(x\, x)\, a)\,(\lambda x.\lambda a.h\,(x\, x)\, a) \tag{9}$$

7.5 Hybrid Applicative Order Reduction to Normal Form

Hybrid applicative order reduction is a hybrid of call-by-value and applicative order reduction. It reduces to normal form, but reduces under lambda abstractions only in argument positions. Therefore the usual call-by-value versions of the recursion combinator, such as (9) above, may be used with this reduction strategy. Thus the hybrid applicative order strategy normalizes more terms than applicative order reduction, while using fewer reduction steps than normal order reduction. The hybrid applicative order strategy relates to call-by-value in the same way that the normal order strategy relates to call-by-name. It resembles Paulson's call-by-value strategy, which works in two phases: first reduce the term by $\xrightarrow{bv}$, then normalize the bodies of any remaining lambda abstractions [16, page 391].

$$\begin{gathered}
x \xrightarrow{ha} x \\
\\
\frac{e \xrightarrow{ha} e'}{(\lambda x.e) \xrightarrow{ha} (\lambda x.e')} \\
\\
\frac{e_1 \xrightarrow{bv} (\lambda x.e) \qquad e_2 \xrightarrow{ha} e'_2 \qquad e[e'_2/x] \xrightarrow{ha} e'}{(e_1\ e_2) \xrightarrow{ha} e'} \\
\\
\frac{e_1 \xrightarrow{bv} e'_1 \not\equiv (\lambda x.e) \qquad e'_1 \xrightarrow{ha} e''_1 \qquad e_2 \xrightarrow{ha} e'_2}{(e_1\ e_2) \xrightarrow{ha} (e''_1\ e'_2)}
\end{gathered} \tag{10}$$

7.6 Head Spine Reduction to Head Normal Form

The head spine strategy performs reductions inside lambda abstractions, but only in head position. This is the reduction strategy implemented by Paulson's `headNF` function [16, page 390].

$$
\begin{array}{c}
x \xrightarrow{he} x \\[2ex]
\dfrac{e \xrightarrow{he} e'}{(\lambda x.e) \xrightarrow{he} (\lambda x.e')} \\[2ex]
\dfrac{e_1 \xrightarrow{he} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{he} e'}{(e_1\ e_2) \xrightarrow{he} e'} \\[2ex]
\dfrac{e_1 \xrightarrow{he} e_1' \not\equiv (\lambda x.e)}{(e_1\ e_2) \xrightarrow{he} (e_1'\ e_2)}
\end{array}
\tag{11}
$$

It is easy to see that the rules generate only head normal form terms. Note that this is not a *head reduction* as defined by Barendregt [4, Definition 8.3.10]: In a (leftmost) head reduction only head redexes are contracted, where a redex $((\lambda x.e_0)\, e_1)$ is a *head redex* if it is preceded to the left only by lambda abstractors of non-redexes, as in $\lambda x_1 \ldots . \lambda x_n . (\lambda x.e_0)\, e_1\ \ldots\ e_m$, with $n \geq 0$ and $m \geq 1$.

To define head reduction, one should use $e_1 \xrightarrow{bn} e_1'$ in the above application rules (11) to avoid premature reduction of inner redexes, similar to the use of $\xrightarrow{bn}$ in the definition of $\xrightarrow{no}$.

7.7 Hybrid Normal Order Reduction to Normal Form

Hybrid normal order reduction is a hybrid of head spine reduction and normal order reduction. It differs from normal order reduction only by reducing the function e_1 in an application to head normal form (by $\xrightarrow{he}$) instead of weak head normal form (by $\xrightarrow{bn}$) before applying it to the argument e_2.

The hybrid normal order strategy resembles Paulson's call-by-name strategy, which works in two phases: first reduce the term by $\xrightarrow{he}$ to head normal form, then normalize unevaluated arguments and bodies of any remaining lambda abstractions [16, page 391].

$$x \xrightarrow{hn} x$$

$$\frac{e \xrightarrow{hn} e'}{(\lambda x.e) \xrightarrow{hn} (\lambda x.e')}$$

$$\frac{e_1 \xrightarrow{he} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{hn} e'}{(e_1\ e_2) \xrightarrow{hn} e'} \tag{12}$$

$$\frac{e_1 \xrightarrow{he} e_1' \not\equiv (\lambda x.e) \qquad e_1' \xrightarrow{hn} e_1'' \qquad e_2 \xrightarrow{hn} e_2'}{(e_1\ e_2) \xrightarrow{hn} (e_1''\ e_2')}$$

These rules generate normal form terms only. The strategy is normalizing, because if the term $(e_1\, e_2)$ has a normal form, then it has a head normal form, and then so has e_1 [4, Proposition 8.3.13].

8 Properties of the Reduction Strategies

The relation defined by each reduction strategy is idempotent. For instance, if $e \xrightarrow{bn} e'$ then $e' \xrightarrow{bn} e'$. To see this, observe that e' is in weak head normal form, so it has form $\lambda x.e''$ or $x\, e_1\ \ldots\ e_n$, where e'' and $e_1, \ldots, e_n$ are arbitrary lambda terms. In the first case, e' reduces to itself by the second rule of (5). In the second case, an induction on n shows that e' reduces to itself by the first and third rule of (5). Similar arguments can be made for the other reduction strategies.

Figure 2 classifies the seven reduction strategies presented in Sections 7.1 to 7.7 according the normal forms (Figure 1) they produce.

	Reduce under abstractions	
Reduce args	**Yes**	**No**
Yes	Normal form **ao**, *no*, *ha*, *ho*	Weak normal form **bv**
No	Head normal form **he**	Weak head normal form **bn**

Fig. 2. Classification of reduction strategies by the normal forms they produce. The 'uniform' reduction strategies are shown in boldface, the 'hybrid' ones in italics.

Inspection of the big-step semantics rules shows that four of the reduction strategies (**ao**, **bn**, **bv**, **he**, shown in bold in Figure 2) are 'uniform': their definition involves only that reduction strategy itself. The remaining three (no, ha, hn) are 'hybrid': each uses one of the 'uniform' strategies for the reduction of the expression e_1 in function position in applications $(e_1\, e_2)$. Figure 3 shows how the 'hybrid' and 'uniform' strategies are related.

Hybrid	Uniform
no	**bn**
ha	**bv**
hn	**he**

Fig. 3. Derivation of hybrid strategies from uniform ones.

9 Tracing: Side-Effecting Substitution, and Contexts

The reducers defined in ML in Section 7 perform the substitutions $e[e_2/x]$ in the same order as prescribed by the operational semantics, thanks to Standard ML semantics: strict evaluation and left-to-right evaluation. But they only return the final reduced lambda term; they do not trace the intermediate steps of the reduction, which is often more interesting from a pedagogical point of view.

ML permits expressions to have side effects, so we can make the substitution function report (e.g. print) the redex just before contracting it. To do this we define a modified substitution function `csubst` which takes as argument another function `c` and applies it to the redex `App(Lam(x, e), ex)` representing $(\lambda x.e)\, e_x$, just before contracting it:

```
fun csubst (c : lam -> unit) ex (Lam(x, e)) =
    (c (App(Lam(x, e), ex));
     subst ex (Lam(x, e)))
```

The function `c : lam -> unit` is evaluated for its side effect only, as shown by the trivial result type `unit`. Evaluating `csubst c ex (Lam(x, e))` has the *effect* of calling `c` on the redex $((\lambda x.e)\, e_x)$, and its *result* is the result of the substitution $e[e_x/x]$, which is the contracted redex.

We could define a function `printlam : lam -> unit` that prints the given lambda term as a side effect. Then replacing the call `subst e2 (Lam(x, e))` in function `cbn` of Section 7.1 by `csubst printlam e2 (Lam(x, e))` will cause the reduction of a term by `cbn` to produce a printed trace of all redexes $((\lambda x.e)\, e_x)$, in the order in which they are contracted.

This still does not give us a usable trace of the evaluation: we do not know where in the current term the printed redex occurs. This is because the function `printlam` is applied only to the redex itself; the term surrounding the redex is implicit. To make the term surrounding the redex explicit, we can use a *context*, a term with a single hole, such as $\lambda x.[\,]$ or $(e_1\,[\,])$ or $([\,]\, e_2)$, where the hole is denoted by $[\,]$. Filling the hole of a context with a lambda term produces a lambda term. The following grammar generates all single-hole contexts:

$$C \;::=\; [\,] \mid \lambda x.C \mid e\,C \mid C\,e \tag{13}$$

A context can be represented by an ML function of type `lam -> lam`. The four forms of contexts (13) can be created using four ML context-building functions:

```
fun id      e  = e
fun Lamx x  e  = Lam(x, e)
fun App2 e1 e2 = App(e1, e2)
fun App1 e2 e1 = App(e1, e2)
```

For instance, `(App1` e_2`)` is the ML function `fn e1 => App(e1, e2)` which represents the context $([\,]\,e_2)$. Filling the hole with the term e_1 is done by computing `(App1` e_2`)` e_1 which evaluates to `App(`e_1`,` e_2`)`, representing the term $(e_1\,e_2)$.

Function composition (`f o g`) composes contexts. For instance, the composition of contexts $\lambda x.[\,]$ and $([\,]\,e_2)$ is `Lamx` x `o App1` e_2, which represents the context $\lambda x.([\,]\,e_2)$. Similarly, the composition of the contexts $([\,]\,e_2)$ and $\lambda x.[\,]$ is `App1` e_2 `o Lamx` x, which represents $((\lambda x.[\,])\,e_2)$.

10 Reduction in Context

To produce a trace of the reduction, we modify the reduction functions defined in Section 7 to take an extra context argument `c` and to use the extended substitution function `csubst`, passing `c` to `csubst`. Then `csubst` will apply `c` to the redex before contracting it. We take the call-by-name reduction function `cbn` (Section 7.1) as an example; the other reduction functions are handled similarly. The reduction function must build up the context `c` as it descends into the term. It does so by composing the context with the appropriate context builder (in this case, only in the `App` branch):

```
fun cbnc c (Var x)       = Var x
  | cbnc c (Lam(x, e))   = Lam(x, e)
  | cbnc c (App(e1, e2)) =
    case cbnc (c o App1 e2) e1 of
        Lam (x, e) => cbnc c (csubst c e2 (Lam(x, e)))
      | e1'        => App(e1', e2)
```

By construction, if c `: lam -> lam` and the evaluation of `cbnc` c e involves a call `cbnc` c' e', then $c[e] \longrightarrow_\beta^* c'[e']$. Also, whenever a call `cbnc` c' $(e_1\,e_2)$ is evaluated, and $e_1 \xrightarrow{bn} (\lambda x.e)$, then function c' is applied to the redex $((\lambda x.e)\,e_2)$ just before it is contracted. Hence a trace of the reduction of term `e` can be obtained just by calling `cbnc` as follows:

```
cbnc printlam e
```

where `printlam : lam -> unit` is a function that prints the lambda term as a side effect. In fact, computing `cbnc printlam (App (App` *add two*`)` *two*`)`, using the encodings from (1), prints the two intermediate terms below. The third term shown is the final result (a weak head normal form):

```
(\m.\n.\f.\x.m f (n f x)) (\f.\x.f (f x)) (\f.\x.f (f x))
(\n.\f.\x.(\f.\x.f (f x)) f (n f x)) (\f.\x.f (f x))
\f.\x.(\f.\x.f (f x)) f ((\f.\x.f (f x)) f x)
```

The trace of a reduction can be defined also by direct instrumentation of the operational semantics (5). Let us define a *trace* to be a finite sequence of lambda terms, denote the empty trace by ϵ, and denote the concatenation of traces s and t by $s \cdot t$. Now we define the relation $e \xrightarrow{bn}{}^{s}_{C} e'$ to mean: under call-by-name, the expression e reduces to e', and if e appears in context C, then s is the trace of the reduction. The trace s will be empty if no redex was contracted in the reduction. If some redex was contracted, the first term in the trace will be e.

The tracing relation corresponding to call-by-name reduction (5) can be defined as shown below:

$$x \xrightarrow{bn}{}^{\epsilon}_{C} x$$

$$(\lambda x.e) \xrightarrow{bn}{}^{\epsilon}_{C} (\lambda x.e)$$

$$\frac{e_1 \xrightarrow{bn}{}^{s}_{C[[\,]\,e_2]} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{bn}{}^{t}_{C} e'}{(e_1\ e_2) \xrightarrow{bn}{}^{s \cdot C[(\lambda x.e)\,e_2] \cdot t}_{C} e'} \tag{14}$$

$$\frac{e_1 \xrightarrow{bn}{}^{s}_{C[[\,]\,e_2]} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{bn}{}^{s}_{C} (e_1'\ e_2)}$$

Thus reduction of a variable x or a lambda abstraction $(\lambda x.e)$ produces the empty trace ϵ. When e_1 reduces to a lambda abstraction, reduction of the application $(e_1\,e_2)$ produces the trace $s \cdot C[(\lambda x.e)\,e_2] \cdot t$, where s traces the reduction of e_1 and t traces the reduction of the contracted redex $e[e_2/x]$.

Tracing versions of the other reduction strategies can be defined analogously.

11 Single-Stepping Reduction

For experimentation it is useful to be able to perform one beta-reduction at a time, or in other words, to single-step the reduction. Again, this can be achieved using side effects in the implementation language. We simply make the context function `c` count the number of redexes contracted (substitutions performed), and set a step limit N before evaluation is started.

When N redexes have been contracted, `c` aborts the reduction by raising an exception `Enough` e', which carries as its argument the term e' that had been obtained after N reductions. An enclosing exception handler returns e' as the result of the reduction. The next invocation of the reduction function simply sets the step limit N one higher, and so on. Thus the reduction of the original term starts over for every new step, but we create the illusion of reducing the term one step at a time.

The main drawback of this approach is that the total time spent performing n steps of reduction is $O(n^2)$. In practice, this does not matter: noboby wants to single-step very long computations.

12 A Web-Based Interface to the Reduction Functions

A web-based interface to the tracing reduction functions can be implemented as an ordinary CGI script. The lambda term to be reduced, the name of the desired reduction strategy, the kind of computation (tracing, single-stepping, etc.) and the step limit are passed as parameters to the script.

Such an implementation has been written in Moscow ML [15] and is available at ⟨http://www.dina.kvl.dk/~sestoft/lamreduce/⟩. The implementation uses the `Mosmlcgi` library to access CGI parameters, and the `Msp` library for efficient structured generation of HTML code.

For tracing, the script uses a function `htmllam : lam -> unit` that prints a lambda term as HTML code, which is then sent to the browser by the web server. Calling `cbnc` (or any other tracing reduction function) with `htmllam` as argument will display a trace of the reduction in the browser.

A trick is used to make the next redex into a hyperlink in the browser. The implementation's representation of lambda terms is extended with labelled subterms, and `csubst` attaches labels $0, 1, \ldots$ to redexes in the order in which they are contracted. When single-stepping a reduction, the last labelled redex inside the term can be formatted as a hyperlink. Clicking on the hyperlink will call the CGI script again to perform one more step of reduction, creating the illusion of single-stepping the reduction as explained above.

13 Conclusion

We have described a simple way to implement lambda calculus reduction, describing reduction strategies using big-step operational semantics, implementing reduction by straightforward reduction functions in Standard ML, and instrumenting them to produce a trace of the reduction, using contexts. This approach is easily extended to other reduction strategies describable by big-step operational semantics.

We find that big-step semantics provides a clear presentation of the reduction strategies, highlighting their differences and making it easy to see what normal forms they produce.

The extension to lazy evaluation, whether using graph reduction or an explicit heap, would be complicated mostly by the need to represent the current term graph or heap, and to print it in a comprehensible way.

The functions for reduction in context were useful for creating a web interface also, running the reduction functions as a CGI script written in ML. The web interface provides a simple platform for students' experiments with lambda calculus encodings and reduction strategies.

Acknowledgements Thanks for Dave Schmidt for helpful comments that have improved contents as well as presentation.

References

1. Abramsky, S., Ong, C.-H.L.: Full Abstraction in the Lazy λ-Calculus. Information and Computation **105**, 2 (1993) 159–268.
2. Ariola, Z.M., Felleisen, M.: The Call-by-Need Lambda Calculus. Journal of Functional Programming **7**, 3 (1997) 265–301.
3. Augustsson, L.: A Compiler for Lazy ML. In: 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas. ACM Press (1984) 218–227.
4. Barendregt, H.P.: The Lambda Calculus. Its Syntax and Semantics. North-Holland (1984).
5. Barendregt, H.P. *et al.*: Needed Reduction and Spine Strategies for the Lambda Calculus. Information and Computation **75** (1987) 191–231.
6. Church, A.: A Note on the Entscheidungsproblem. Journal of Symbolic Logic **1** (1936) 40–41, 101–102.
7. Engelfriet, J., Schmidt, E.M.: IO and OI. Journal of Computer and System Sciences **15** (1977) 328–353 and **16** (1978) 67–99.
8. Felleisen, M., Hieb, R.: The Revised Report on the Syntactic Theories of Sequential Control and State. Theoretical Computer Science **103**, 2 (1992) 235–271.
9. Kahn, G.: Natural Semantics. In: STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany. Lecture Notes in Computer Science, Vol. 247. Springer-Verlag (1987) 22–39.
10. Landin, P.J.: The Mechanical Evaluation of Expressions. Computer Journal **6**, 4 (1964) 308–320.
11. Landin, P.J.: A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. Communications of the ACM **8**, 2 (1965) 89–101.
12. Landin, P.J.: The Next 700 Programming Languages. Communications of the ACM **9**, 3 (1966) 157–166.
13. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In: Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993. ACM Press (1993) 144–154.
14. Milner, R., Tofte, M., Harper, R., MacQueen, D.B.: The Definition of Standard ML (Revised). The MIT Press (1997).
15. Moscow ML is available at ⟨http://www.dina.kvl.dk/~sestoft/mosml.html⟩.
16. Paulson, L.C.: ML for the Working Programmer. Second edition. Cambridge University Press (1996).
17. Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall (1987).
18. Peyton Jones, S.L., Hughes, J. (eds.): Haskell 98: A Non-Strict, Purely Functional Language. At ⟨http://www.haskell.org/onlinereport/⟩.
19. Plotkin, G.: Call-by-Name, Call-by-Value and the λ-Calculus. Theoretical Computer Science **1** (1975) 125–159.
20. Revised^4 Report on the Algorithmic Language Scheme, IEEE Std 1178-1990. Institute of Electrical and Electronic Engineers (1991).
21. Sestoft, P.: Deriving a Lazy Abstract Machine. Journal of Functional Programming **7**, 3 (1997) 231–264.
22. Stoy, J.E.: The Scott-Strachey Approach to Programming Language Theory. The MIT Press (1977).
23. Turner, D.A.: A New Implementation Technique for Applicative Languages. Software – Practice and Experience **9** (1979) 31–49.
24. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus. D.Phil. thesis, Oxford University, September 1971.

J. Functional Programming **9** (4): 355–372, July 1999. Printed in the United Kingdom

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON
University of Nottingham, Nottingham, UK
http://www.cs.nott.ac.uk/~gmh

Abstract

In functional programming, *fold* is a standard operator that encapsulates a simple pattern of recursion for processing lists. This article is a tutorial on two key aspects of the fold operator for lists. First of all, we emphasize the use of the *universal property* of fold both as a *proof principle* that avoids the need for inductive proofs, and as a *definition principle* that guides the transformation of recursive functions into definitions using fold. Secondly, we show that even though the pattern of recursion encapsulated by fold is simple, in a language with tuples and functions as first-class values the fold operator has greater *expressive power* than might first be expected.

Capsule Review

Within the last ten to fifteen years, the algebra of datatypes has become a stable and well understood element of the mathematics of program construction. Graham Hutton's paper is a highly readable, elementary introduction to the algebra centred on the well-known function on lists. The paper distinguishes itself by focusing on how the properties are used for the crucial task of 'constructing' programs, rather than on the post hoc verification of existing programs. Several well-chosen examples are given, beginning at an elementary level and progressing to more advanced applications. The paper concludes with a good overview and bibliography of recent literature which develops the theory and its applications in more depth.

1 Introduction

Many programs that involve repetition are naturally expressed using some form of recursion, and properties proved of such programs using some form of induction. Indeed, in the functional approach to programming, recursion and induction are the primary tools for defining and proving properties of programs.

Not surprisingly, many recursive programs will share a common pattern of recursion, and many inductive proofs will share a common pattern of induction. Repeating the same patterns again and again is tedious, time consuming, and prone to error. Such repetition can be avoided by introducing special *recursion operators* and *proof principles* that encapsulate the common patterns, allowing us to concentrate on the parts that are different for each application.

In functional programming, *fold* (also known as *foldr*) is a standard recursion

operator that encapsulates a common pattern of recursion for processing lists. The fold operator comes equipped with a proof principle called *universality*, which encapsulates a common pattern of inductive proof concerning lists. Fold and its universal property together form the basis of a simple but powerful calculational theory of programs that process lists. This theory generalises from lists to a variety of other datatypes, but for simplicity we restrict our attention to lists.

This article is a tutorial on two key aspects of the fold operator for lists. First of all, we emphasize the use of the universal property of fold (together with the derived *fusion property*) both as *proof principles* that avoid the need for inductive proofs, and as *definition principles* that guide the transformation of recursive functions into definitions using fold. Secondly, we show that even though the pattern of recursion encapsulated by fold is simple, in a language with tuples and functions as first-class values the fold operator has greater *expressive power* than might first be expected, thus permitting the powerful universal and fusion properties of fold to be applied to a larger class of programs. The article concludes with a survey of other work on recursion operators that we do not have space to pursue here.

The article is aimed at a reader who is familiar with the basics of functional programming, say to the level of Bird and Wadler (1988) and Bird (1998). All programs in the article are written in Haskell (Peterson *et al.*, 1997), the standard lazy functional programming language. However, no special features of Haskell are used, and the ideas can easily be adapted to other functional languages.

2 The fold operator

The *fold* operator has its origins in recursion theory (Kleene, 1952), while the use of *fold* as a central concept in a programming language dates back to the reduction operator of APL (Iverson, 1962), and later to the insertion operator of FP (Backus, 1978). In Haskell, the *fold* operator for lists can be defined as follows:

$$\begin{array}{lcl} fold & :: & (\alpha \to \beta \to \beta) \to \beta \to ([\alpha] \to \beta) \\ fold\ f\ v\ [\,] & = & v \\ fold\ f\ v\ (x : xs) & = & f\ x\ (fold\ f\ v\ xs) \end{array}$$

That is, given a function f of type $\alpha \to \beta \to \beta$ and a value v of type β, the function $fold\ f\ v$ processes a list of type $[\alpha]$ to give a value of type β by replacing the *nil* constructor $[\,]$ at the end of the list by the value v, and each *cons* constructor $(:)$ within the list by the function f. In this manner, the *fold* operator encapsulates a simple pattern of recursion for processing lists, in which the two constructors for lists are simply replaced by other values and functions. A number of familiar functions on lists have a simple definition using *fold*. For example:

$$\begin{array}{lcl} sum & :: & [Int] \to Int \\ sum & = & fold\ (+)\ 0 \end{array} \qquad \begin{array}{lcl} product & :: & [Int] \to Int \\ product & = & fold\ (\times)\ 1 \end{array}$$

$$\begin{array}{lcl} and & :: & [Bool] \to Bool \\ and & = & fold\ (\wedge)\ True \end{array} \qquad \begin{array}{lcl} or & :: & [Bool] \to Bool \\ or & = & fold\ (\vee)\ False \end{array}$$

Recall that enclosing an infix operator $\oplus$ in parentheses $(\oplus)$ converts the operator into a prefix function. This notational device, called *sectioning*, is often useful when defining simple functions using *fold*. If required, one of the arguments to the operator can also be enclosed in the parentheses. For example, the function $(+\!+)$ that appends two lists to give a single list can be defined as follows:

$$\begin{array}{lcl} (+\!+) & :: & [\alpha] \to [\alpha] \to [\alpha] \\ (+\!+\ ys) & = & fold\ (:)\ ys \end{array}$$

In all our examples so far, the constructor $(:)$ is replaced by a built-in function. However, in most applications of *fold* the constructor $(:)$ will be replaced by a user-defined function, often defined as a nameless function using the λ notation, as in the following definitions of standard list-processing functions:

$$\begin{array}{lcl} length & :: & [\alpha] \to Int \\ length & = & fold\ (\lambda x\ n \to 1 + n)\ 0 \end{array}$$

$$\begin{array}{lcl} reverse & :: & [\alpha] \to [\alpha] \\ reverse & = & fold\ (\lambda x\ xs \to xs +\!+ [x])\ [\,] \end{array}$$

$$\begin{array}{lcl} map & :: & (\alpha \to \beta) \to ([\alpha] \to [\beta]) \\ map\ f & = & fold\ (\lambda x\ xs \to f\ x\ :\ xs)\ [\,] \end{array}$$

$$\begin{array}{lcl} filter & :: & (\alpha \to Bool) \to ([\alpha] \to [\alpha]) \\ filter\ p & = & fold\ (\lambda x\ xs \to \textbf{if}\ \ p\ \ x\ \ \textbf{then}\ \ x : xs\ \ \textbf{else}\ \ xs)\ \ [\,] \end{array}$$

Programs written using *fold* can be less readable than programs written using explicit recursion, but can be constructed in a systematic manner, and are better suited to transformation and proof. For example, we will see later on in the article how the above definition for *map* using *fold* can be constructed from the standard definition using explicit recursion, and more importantly, how the definition using *fold* simplifies the process of proving properties of the *map* function.

3 The universal property of fold

As with the *fold* operator itself, the universal property of *fold* also has its origins in recursion theory. The first systematic use of the universal property in functional programming was by Malcolm (1990*a*), in his generalisation of Bird and Meerten's theory of lists (Bird, 1989; Meertens, 1983) to arbitrary regular datatypes. For finite lists, the universal property of *fold* can be stated as the following equivalence between two definitions for a function g that processes lists:

$$\begin{array}{lcl} g\ \ [\,] & = & v \\ g\ \ (x : xs) & = & f\ \ x\ \ (g\ \ xs) \end{array} \qquad \Leftrightarrow \qquad g\ \ =\ \ fold\ \ f\ \ v$$

In the right-to-left direction, substituting $g = fold\ \ f\ \ v$ into the two equations for g gives the recursive definition for *fold*. Conversely, in the left-to-right direction the two equations for g are precisely the assumptions required to show that $g = fold\ \ f\ \ v$

using a simple proof by induction on finite lists (Bird, 1998). Taken as a whole, the universal property states that for finite lists the function $fold\ f\ v$ is not just a solution to its defining equations, but in fact the *unique* solution.

The key to the utility of the universal property is that it makes explicit the two assumptions required for a certain pattern of inductive proof. For specific cases then, by verifying the two assumptions (which can typically be done without the need for induction) we can then appeal to the universal property to complete the inductive proof that $g = fold\ f\ v$. In this manner, the universal property of *fold* encapsulates a simple pattern of inductive proof concerning lists, just as the *fold* operator itself encapsulates a simple pattern of recursion for processing lists.

The universal property of *fold* can be generalised to handle partial and infinite lists (Bird, 1998), but for simplicity we only consider finite lists in this article.

3.1 Universality as a proof principle

The primary application of the universal property of *fold* is as a proof principle that avoids the need for inductive proofs. As a simple first example, consider the following equation between functions that process a list of numbers:

$$(+1) \cdot sum \;=\; fold\ (+)\ 1$$

The left-hand function sums a list and then increments the result. The right-hand function processes a list by replacing each $(:)$ by the addition function $(+)$ and the empty list $[\,]$ by the constant 1. The equation asserts that these two functions always give the same result when applied to the same list.

To prove the above equation, we begin by observing that it matches the right-hand side $g = fold\ f\ v$ of the universal property of *fold*, with $g = (+1) \cdot sum$, $f = (+)$, and $v = 1$. Hence, by appealing to the universal property, we conclude that the equation to be proved is equivalent to the following two equations:

$$\begin{array}{lcl} ((+1) \cdot sum)\ [\,] & = & 1 \\ ((+1) \cdot sum)\ (x : xs) & = & (+)\ x\ (((+1) \cdot sum)\ xs) \end{array}$$

At first sight, these may seem more complicated than the original equation. However, simplifying using the definitions of composition and sectioning gives

$$\begin{array}{lcl} sum\ [\,] + 1 & = & 1 \\ sum\ (x : xs) + 1 & = & x + (sum\ xs + 1) \end{array}$$

which can now be verified by simple calculations, shown here in two columns:

$$\begin{array}{ll} & sum\ [\,] + 1 \\ = & \quad \{ \text{ Definition of } sum \} \\ & 0 + 1 \\ = & \quad \{ \text{ Arithmetic} \} \\ & 1 \end{array} \qquad \begin{array}{ll} & sum\ (x : xs) + 1 \\ = & \quad \{ \text{ Definition of } sum \} \\ & (x + sum\ xs) + 1 \\ = & \quad \{ \text{ Arithmetic} \} \\ & x + (sum\ xs + 1) \end{array}$$

This completes the proof. Normally this proof would have required an explicit use of induction. However, in the above proof the use of induction has been encapsulated

in the universal property of *fold*, with the result that the proof is reduced to a simplification step followed by two simple calculations.

In general, any two functions on lists that can be proved equal by induction can also be proved equal using the universal property of the *fold* operator, provided, of course, that the functions can be expressed using *fold*. The expressive power of the *fold* operator will be addressed later on in the article.

3.2 The fusion property of fold

Now let us generalise from the *sum* example and consider the following equation between functions that process a list of values:

$$h \cdot fold\ g\ w\ =\ fold\ f\ v$$

This pattern of equation occurs frequently when reasoning about programs written using *fold*. It is not true in general, but we can use the universal property of *fold* to calculate conditions under which the equation will indeed be true. The equation matches the right-hand side of the universal property, from which we conclude that the equation is equivalent to the following two equations:

$$\begin{array}{lcl} (h \cdot fold\ g\ w)\ [\,] & = & v \\ (h \cdot fold\ g\ w)\ (x : xs) & = & f\ x\ ((h \cdot fold\ g\ w)\ xs) \end{array}$$

Simplifying using the definition of composition gives

$$\begin{array}{lcl} h\ (fold\ g\ w\ [\,]) & = & v \\ h\ (fold\ g\ w\ (x : xs)) & = & f\ x\ (h\ (fold\ g\ w\ xs)) \end{array}$$

which can now be further simplified by two calculations:

$$\begin{array}{ll} & h\ (fold\ g\ w\ [\,])\ =\ v \\ \Leftrightarrow & \quad \{\ \text{Definition of } fold\ \} \\ & h\ w\ =\ v \end{array}$$

and

$$\begin{array}{ll} & h\ (fold\ g\ w\ (x : xs))\ =\ f\ x\ (h\ (fold\ g\ w\ xs)) \\ \Leftrightarrow & \quad \{\ \text{Definition of } fold\ \} \\ & h\ (g\ x\ (fold\ g\ w\ xs))\ =\ f\ x\ (h\ (fold\ g\ w\ xs)) \\ \Leftarrow & \quad \{\ \text{Generalising } (fold\ g\ w\ xs) \text{ to a fresh variable } y\ \} \\ & h\ (g\ x\ y)\ =\ f\ x\ (h\ y) \end{array}$$

That is, using the universal property of *fold* we have calculated – without an explicit use of induction – two simple conditions that are together sufficient to ensure for all finite lists that the composition of an arbitrary function and a *fold* can be fused together to give a single *fold*. Following this interpretation, this property is called the *fusion* property of the *fold* operator, and can be stated as follows:

$$\begin{array}{lcl} h\ w & = & v \\ h\ (g\ x\ y) & = & f\ x\ (h\ y) \end{array} \quad \Rightarrow \quad h \cdot fold\ g\ w\ =\ fold\ f\ v$$

The first systematic use of the fusion property in functional programming was again by Malcolm (1990*a*), generalising earlier work by Bird (1989) and Meertens (1983). As with the universal property, the primary application of the fusion property is as a proof principle that avoids the need for inductive proofs. In fact, for many practical examples the fusion property is often preferable to the universal property. As a simple first example, consider again the equation:

$$(+1) \cdot sum \;=\; fold\ (+)\ 1$$

In the previous section this equation was proved using the universal property of *fold*. However, the proof is simpler using the fusion property. First, we replace the function *sum* by its definition using *fold* given earlier:

$$(+1) \cdot fold\ (+)\ 0 \;=\; fold\ (+)\ 1$$

The equation now matches the conclusion of the fusion property, from which we conclude that the equation follows from the following two assumptions:

$$\begin{array}{lcl} (+1)\ 0 & = & 1 \\ (+1)\ ((+)\ x\ y) & = & (+)\ x\ ((+1)\ y) \end{array}$$

Simplifying these equations using the definition of sectioning gives $0 + 1 = 1$ and $(x + y) + 1 = x + (y + 1)$, which are true by simple properties of arithmetic. More generally, by replacing the use of addition in this example by an arbitrary infix operator $\oplus$ that is associative, a simple application of fusion shows that:

$$(\oplus\ a) \cdot fold\ (\oplus)\ b \;=\; fold\ (\oplus)\ (b \oplus a)$$

For a more interesting example, consider the following well-known equation, which asserts that the *map* operator distributes over function composition $(\cdot)$:

$$map\ f \cdot map\ g \;=\; map\ (f \cdot g)$$

By replacing the second and third occurrences of the *map* operator in the equation by its definition using *fold* given earlier, the equation can be rewritten in a form that matches the conclusion of the fusion property:

$$\begin{array}{c} map\ f \cdot fold\ (\lambda x\ xs \rightarrow g\ x\ :\ xs)\ [\,] \\ = \\ fold\ (\lambda x\ xs \rightarrow (f \cdot g)\ x\ :\ xs)\ [\,] \end{array}$$

Appealing to the fusion property and then simplifying gives the following two equations, which are trivially true by the definitions of *map* and $(\cdot)$:

$$\begin{array}{lcl} map\ f\ [\,] & = & [\,] \\ map\ f\ (g\ x\ :\ y) & = & (f \cdot g)\ x\ :\ map\ f\ y \end{array}$$

In addition to the fusion property, there are a number of other useful properties of the *fold* operator that can be derived from the universal property (Bird, 1998). However, the fusion property suffices for many practical cases, and one can always revert to the full power of the universal property if fusion is not appropriate.

3.3 Universality as a definition principle

As well as being used as a proof principle, the universal property of *fold* can also be used as a definition principle that guides the transformation of recursive functions into definitions using *fold*. As a simple first example, consider the recursively defined function *sum* that calculates the sum of a list of numbers:

$$\begin{array}{lcl} sum & :: & [Int] \to Int \\ sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array}$$

Suppose now that we want to redefine *sum* using *fold*. That is, we want to solve the equation $sum = fold\ f\ v$ for a function f and a value v. We begin by observing that the equation matches the right-hand side of the universal property, from which we conclude that the equation is equivalent to the following two equations:

$$\begin{array}{lcl} sum\ [\,] & = & v \\ sum\ (x : xs) & = & f\ x\ (sum\ xs) \end{array}$$

From the first equation and the definition of *sum*, it is immediate that $v = 0$. From the second equation, we calculate a definition for f as follows:

$$\begin{array}{ll} & sum\ (x : xs) = f\ x\ (sum\ xs) \\ \Leftrightarrow & \quad \{\ \text{Definition of } sum\ \} \\ & x + sum\ xs = f\ x\ (sum\ xs) \\ \Leftarrow & \quad \{\ \dagger\ \text{Generalising } (sum\ xs) \text{ to } y\ \} \\ & x + y = f\ x\ y \\ \Leftrightarrow & \quad \{\ \text{Functions}\ \} \\ & f = (+) \end{array}$$

That is, using the universal property we have calculated that:

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

Note that the key step (†) above in calculating a definition for f is the generalisation of the expression $sum\ xs$ to a fresh variable y. In fact, such a generalisation step is not specific to the *sum* function, but will be a key step in the transformation of any recursive function into a definition using *fold* in this manner.

Of course, the *sum* example above is rather artificial, because the definition of *sum* using *fold* is immediate. However, there are many examples of functions whose definition using *fold* is not so immediate. For example, consider the recursively defined function *map f* that applies a function f to each element of a list:

$$\begin{array}{lcl} map & :: & (\alpha \to \beta) \to ([\alpha] \to [\beta]) \\ map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x : map\ f\ xs \end{array}$$

To redefine *map f* using *fold* we must solve the equation $map\ f = fold\ g\ v$ for a function g and a value v. By appealing to the universal property, we conclude that this equation is equivalent to the following two equations:

$$\begin{array}{lcl} map\ f\ [\,] & = & v \\ map\ f\ (x : xs) & = & g\ x\ (map\ f\ xs) \end{array}$$

From the first equation and the definition of *map* it is immediate that $v = [\,]$. From the second equation, we calculate a definition for g as follows:

$$\begin{array}{ll} & map\ f\ (x : xs) = g\ x\ (map\ f\ xs) \\ \Leftrightarrow & \quad \{ \text{ Definition of } map \text{ } \} \\ & f\ x : map\ f\ xs = g\ x\ (map\ f\ xs) \\ \Leftarrow & \quad \{ \text{ Generalising } (map\ f\ xs) \text{ to } ys \text{ } \} \\ & f\ x : ys = g\ x\ ys \\ \Leftrightarrow & \quad \{ \text{ Functions } \} \\ & g = \lambda x\ ys \rightarrow f\ x : ys \end{array}$$

That is, using the universal property we have calculated that:

$$map\ f \quad = \quad fold\ (\lambda x\ ys \rightarrow f\ x : ys)\ [\,]$$

In general, any function on lists that can be expressed using the *fold* operator can be transformed into such a definition using the universal property of *fold*.

4 Increasing the power of fold: generating tuples

As a simple first example of the use of *fold* to generate tuples, consider the function *sumlength* that calculates the sum and length of a list of numbers:

$$\begin{array}{lcl} sumlength & :: & [Int] \rightarrow (Int, Int) \\ sumlength\ xs & = & (sum\ xs,\ length\ xs) \end{array}$$

By a straightforward combination of the definitions of the functions *sum* and *length* using *fold* given earlier, the function *sumlength* can be redefined as a single application of *fold* that generates a pair of numbers from a list of numbers:

$$sumlength \quad = \quad fold\ (\lambda n\ (x, y) \rightarrow (n + x,\ 1 + y))\ (0, 0)$$

This definition is more efficient than the original definition, because it only makes a single traversal over the argument list, rather than two separate traversals. Generalising from this example, any pair of applications of *fold* to the same list can always be combined to give a single application of *fold* that generates a pair, by appealing to the so-called 'banana split' property of *fold* (Meijer, 1992). The strange name of this property derives from the fact that the *fold* operator is sometimes written using brackets $(\!|\ |\!)$ that resemble bananas, and the pairing operator is sometimes called split. Hence, their combination can be termed a banana split!

As a more interesting example, let us consider the function *dropWhile* p that removes initial elements from a list while all the elements satisfy the predicate p:

$$\begin{array}{lcl} dropWhile & :: & (\alpha \rightarrow Bool) \rightarrow ([\alpha] \rightarrow [\alpha]) \\ dropWhile\ p\ [\,] & = & [\,] \\ dropWhile\ p\ (x : xs) & = & \textbf{if}\ p\ x\ \textbf{then}\ dropWhile\ p\ xs\ \textbf{else}\ x : xs \end{array}$$

Suppose now that we want to redefine $dropWhile\ p$ using the *fold* operator. By appealing to the universal property, we conclude that the equation $dropWhile\ p = fold\ f\ v$ is equivalent to the following two equations:

$$\begin{array}{lcl} dropWhile\ p\ [\,] & = & v \\ dropWhile\ p\ (x : xs) & = & f\ x\ (dropWhile\ p\ xs) \end{array}$$

¿From the first equation it is immediate that $v = [\,]$. From the second equation, we attempt to calculate a definition for f in the normal manner:

$$\begin{array}{ll} & dropWhile\ p\ (x : xs) = f\ x\ (dropWhile\ p\ xs) \\ \Leftrightarrow & \quad \{\ \text{Definition of } dropWhile\ \} \\ & \textbf{if}\ p\ x\ \textbf{then}\ dropWhile\ p\ xs\ \textbf{else}\ x : xs = f\ x\ (dropWhile\ p\ xs) \\ \Leftarrow & \quad \{\ \text{Generalising } (dropWhile\ p\ xs) \text{ to } ys\ \} \\ & \textbf{if}\ p\ x\ \textbf{then}\ ys\ \textbf{else}\ x : xs = f\ x\ ys \end{array}$$

Unfortunately, the final line above is not a valid definition for f, because the variable xs occurs freely. In fact, it is not possible to redefine $dropWhile\ p$ directly using *fold*. However, it is possible indirectly, because the more general function

$$\begin{array}{lcl} dropWhile' & :: & (\alpha \to Bool) \to ([\alpha] \to ([\alpha],\ [\alpha])) \\ dropWhile'\ p\ xs & = & (dropWhile\ p\ xs,\ xs) \end{array}$$

that pairs up the result of applying $dropWhile\ p$ to a list with the list itself *can* be redefined using *fold*. By appealing to the universal property, we conclude that the equation $dropWhile'\ p = fold\ f\ v$ is equivalent to the following two equations:

$$\begin{array}{lcl} dropWhile'\ p\ [\,] & = & v \\ dropWhile'\ p\ (x : xs) & = & f\ x\ (dropWhile'\ p\ xs) \end{array}$$

A simple calculation from the first equation gives $v = ([\,], [\,])$. From the second equation, we calculate a definition for f as follows:

$$\begin{array}{ll} & dropWhile'\ p\ (x : xs) = f\ x\ (dropWhile'\ p\ xs) \\ \Leftrightarrow & \quad \{\ \text{Definition of } dropWhile'\ \} \\ & (dropWhile\ p\ (x : xs),\ x : xs) = f\ x\ (dropWhile\ p\ xs,\ xs) \\ \Leftrightarrow & \quad \{\ \text{Definition of } dropWhile\ \} \\ & (\textbf{if}\ p\ x\ \textbf{then}\ dropWhile\ p\ xs\ \textbf{else}\ x : xs,\ x : xs) \\ & \quad = f\ x\ (dropWhile\ p\ xs,\ xs) \\ \Leftarrow & \quad \{\ \text{Generalising } (dropWhile\ p\ xs) \text{ to } ys\ \} \\ & (\textbf{if}\ p\ x\ \textbf{then}\ ys\ \textbf{else}\ x : xs,\ x : xs) = f\ x\ (ys, xs) \end{array}$$

Note that the final line above *is* a valid definition for f, because all the variables are bound. In summary, using the universal property we have calculated that:

$$\begin{array}{l} dropWhile'\ p = fold\ f\ v \\ \quad \textbf{where} \\ \quad\quad f\ x\ (ys, xs) = (\textbf{if}\ p\ x\ \textbf{then}\ ys\ \textbf{else}\ x : xs,\ x : xs) \\ \quad\quad v = ([\,], [\,]) \end{array}$$

This definition satisfies the equation $dropWhile'\ p\ xs = (dropWhile\ p\ xs,\ xs)$, but does not make use of *dropWhile* in its definition. Hence, the function *dropWhile* itself can now be redefined simply by $dropWhile\ p = fst \cdot dropWhile'\ p$.

In conclusion, by first generalising to a function *dropWhile′* that pairs the desired result with the argument list, we have now shown how the function *dropWhile* can be redefined in terms of *fold*, as required. In fact, this result is an instance of a general theorem (Meertens, 1992) that states that any function on finite lists that is defined by pairing the desired result with the argument list can always be redefined in terms of *fold*, although not always in a way that does not make use of the original (possibly recursive) definition for the function.

4.1 Primitive recursion

In this section we show that by using the tupling technique from the previous section, every primitive recursive function on lists can be redefined in terms of *fold*. Let us begin by recalling that the *fold* operator captures the following simple pattern of recursion for defining a function h that processes lists:

$$\begin{array}{lcl} h\ [\,] & = & v \\ h\ (x : xs) & = & g\ x\ (h\ xs) \end{array}$$

Such functions can be redefined by $h = fold\ g\ v$. We will generalise this pattern of recursion to primitive recursion in two steps. First of all, we introduce an extra argument y to the function h, which in the base case is processed by a new function f, and in the recursive case is passed unchanged to the functions g and h. That is, we now consider the following pattern of recursion for defining a function h:

$$\begin{array}{lcl} h\ y\ [\,] & = & f\ y \\ h\ y\ (x : xs) & = & g\ y\ x\ (h\ y\ xs) \end{array}$$

By simple observation, or a routine application of the universal property of *fold*, the function $h\ y$ can be redefined using *fold* as follows:

$$h\ y = fold\ (g\ y)\ (f\ y)$$

For the second step, we introduce the list xs as an extra argument to the auxiliary function g. That is, we now consider the following pattern for defining h:

$$\begin{array}{lcl} h\ y\ [\,] & = & f\ y \\ h\ y\ (x : xs) & = & g\ y\ x\ xs\ (h\ y\ xs) \end{array}$$

This pattern of recursion on lists is called *primitive recursion* (Kleene, 1952). Technically, the standard definition of primitive recursion requires that the argument y is a finite sequence of arguments. However, because tuples are first-class values in Haskell, treating the case of a single argument y is sufficient.

In order to redefine primitive recursive functions in terms of *fold*, we must solve the equation $h\ y = fold\ i\ j$ for a function i and a value j. This is not possible directly, but is possible indirectly, because the more general function

$$k\ y\ xs = (h\ y\ xs,\ xs)$$

that pairs up the result of applying $h\ y$ to a list with the list itself *can* be redefined using *fold*. By appealing to the universal property of *fold*, we conclude that the equation $k\ y = fold\ i\ j$ is equivalent to the following two equations:

$$\begin{array}{lcl} k\ y\ [\,] & = & j \\ k\ y\ (x : xs) & = & i\ x\ (k\ y\ xs) \end{array}$$

A simple calculation from the first equation gives $j = (f\ y,\ [\,])$. ¿From the second equation, we calculate a definition for i as follows:

$$\begin{array}{ll} & k\ y\ (x : xs) = i\ x\ (k\ y\ xs) \\ \Leftrightarrow & \quad \{ \text{ Definition of } k \ \} \\ & (h\ y\ (x : xs),\ x : xs) = i\ x\ (h\ y\ xs,\ xs) \\ \Leftrightarrow & \quad \{ \text{ Definition of } h \ \} \\ & (g\ y\ x\ xs\ (h\ y\ xs),\ x : xs) = i\ x\ (h\ y\ xs,\ xs) \\ \Leftarrow & \quad \{ \text{ Generalising } (h\ y\ xs) \text{ to } z \ \} \\ & (g\ y\ x\ xs\ z,\ x : xs) = i\ x\ (z,\ xs) \end{array}$$

In summary, using the universal property we have calculated that:

$$\begin{array}{lcl} k\ y & = & fold\ i\ j \\ & & \textbf{where} \\ & & \quad \begin{array}{lcl} i\ x\ (z,\ xs) & = & (g\ y\ x\ xs\ z,\ x : xs) \\ j & = & (f\ y,\ [\,]) \end{array} \end{array}$$

This definition satisfies the equation $k\ y\ xs = (h\ y\ xs,\ xs)$, but does not make use of h in its definition. Hence, the primitive recursive function h itself can now be redefined simply by $h\ y = fst \cdot k\ y$. In conclusion, we have now shown how an arbitrary primitive recursive function on lists can be redefined in terms of *fold*.

Note that the use of tupling to define primitive recursive functions in terms of *fold* is precisely the key to defining the predecessor function for the Church numerals (Barendregt, 1984). Indeed, the intuition behind the representation of the natural numbers (or more generally, any inductive datatype) in the λ-calculus is the idea of representing each number by its *fold* operator. For example, the number $3 = succ\ (succ\ (succ\ zero))$ is represented by the term $\lambda f\ x \rightarrow f\ (f\ (f\ x))$, which is the *fold* operator for 3 in the sense that the arguments f and x can be viewed as the replacements for the *succ* and *zero* constructors respectively.

5 Using fold to generate functions

Having functions as first-class values increases the power of primitive recursion, and hence the power of the *fold* operator. As a simple first example of the use of *fold* to generate functions, the function *compose* that forms the composition of a list of functions can be defined using *fold* by replacing each $(:)$ in the list by the composition function $(\cdot)$, and the empty list $[\,]$ by the identity function *id*:

$$\begin{array}{lcl} compose & :: & [\alpha \rightarrow \alpha] \rightarrow (\alpha \rightarrow \alpha) \\ compose & = & fold\ (\cdot)\ id \end{array}$$

As a more interesting example, let us consider the problem of summing a list of numbers. The natural definition for such a function, $sum = fold\ (+)\ 0$, processes the numbers in the list in right-to-left order. However, it is also possible to define a function *suml* that processes the numbers in left-to-right order. The *suml* function is naturally defined using an auxiliary function $suml'$ that is itself defined by explicit recursion and makes use of an accumulating parameter n:

$$\begin{array}{lcl} suml & :: & [Int] \to Int \\ suml\ xs & = & suml'\ xs\ 0 \\ & & \textbf{where} \\ & & \quad suml'\ [\,]\ n = n \\ & & \quad suml'\ (x : xs)\ n = suml'\ xs\ (n + x) \end{array}$$

Because the addition function $(+)$ is associative and the constant 0 is unit for addition, the functions *suml* and *sum* always give the same result when applied to the same list. However, the function *suml* has the potential to be more efficient, because it can easily be modified to run in constant space (Bird, 1998).

Suppose now that we want to redefine *suml* using the *fold* operator. This is not possible directly, but is possible indirectly, because the auxiliary function

$$suml' :: [Int] \to (Int \to Int)$$

can be redefined using *fold*. By appealing to the universal property, we conclude that the equation $suml' = fold\ f\ v$ is equivalent to the following two equations:

$$\begin{array}{lcl} suml'\ [\,] & = & v \\ suml'\ (x : xs) & = & f\ x\ (suml'\ xs) \end{array}$$

A simple calculation from the first equation gives $v = id$. From the second equation, we calculate a definition for the function f as follows:

$$\begin{array}{ll} & suml'\ (x : xs) = f\ x\ (suml'\ xs) \\ \Leftrightarrow & \quad \{\ \text{Functions}\ \} \\ & suml'\ (x : xs)\ n = f\ x\ (suml'\ xs)\ n \\ \Leftrightarrow & \quad \{\ \text{Definition of } suml'\ \} \\ & suml'\ xs\ (n + x) = f\ x\ (suml'\ xs)\ n \\ \Leftarrow & \quad \{\ \text{Generalising } (suml'\ xs) \text{ to } g\ \} \\ & g\ (n + x) = f\ x\ g\ n \\ \Leftrightarrow & \quad \{\ \text{Functions}\ \} \\ & f = \lambda x\ g \to (\lambda n \to g\ (n + x)) \end{array}$$

In summary, using the universal property we have calculated that:

$$suml' = fold\ (\lambda x\ g \to (\lambda n \to g\ (n + x)))\ id$$

This definition states that $suml'$ processes a list by replacing the empty list $[\,]$ by the identity function *id* on lists, and each constructor $(:)$ by the function that takes a number x and a function g, and returns the function that takes an accumulator value n and returns the result of applying g to the new accumulator value $n + x$.

Note that the structuring of the arguments to $suml' :: [Int] \to (Int \to Int)$ is crucial to its definition using *fold*. In particular, if the order of the two arguments is

swapped or they are supplied as a pair, then the type of *suml′* means that it can no longer be defined directly using *fold*. In general, some care regarding the structuring of arguments is required when aiming to redefine functions using *fold*. Moreover, at first sight one might imagine that *fold* can only be used to define functions that process the elements of lists in right-to-left order. However, as the definition of *suml′* using *fold* shows, the order in which the elements are processed depends on the arguments of *fold*, not on *fold* itself.

In conclusion, by first redefining the auxiliary function *suml′* using *fold*, we have now shown how the function *suml* can be redefined in terms of *fold*, as required:

$$suml\ xs \quad = \quad fold\ (\lambda x\ g \rightarrow (\lambda n \rightarrow g\ (n + x)))\ id\ xs\ 0$$

We end this section by remarking that the use of *fold* to generate functions provides an elegant technique for the implementation of 'attribute grammars' in functional languages (Fokkinga *et al.*, 1991; Swierstra *et al.*, 1998).

5.1 The foldl operator

Now let us generalise from the *suml* example and consider the standard operator *foldl* that processes the elements of a list in left-to-right order by using a function f to combine values, and a value v as the starting value:

$$\begin{array}{lcl} foldl & :: & (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow ([\alpha] \rightarrow \beta) \\ foldl\ f\ v\ [\,] & = & v \\ foldl\ f\ v\ (x : xs) & = & foldl\ f\ (f\ v\ x)\ xs \end{array}$$

Using this operator, *suml* can be redefined simply by $suml = foldl\ (+)\ 0$. Many other functions can be defined in a simple way using *foldl*. For example, the standard function *reverse* can redefined using *foldl* as follows:

$$\begin{array}{lcl} reverse & :: & [\alpha] \rightarrow [\alpha] \\ reverse & = & foldl\ (\lambda xs\ x \rightarrow x : xs)\ [\,] \end{array}$$

This definition is more efficient than our original definition using *fold*, because it avoids the use of the inefficient append operator (++) for lists.

A simple generalisation of the calculation in the previous section for the function *suml* shows how to redefine the function *foldl* in terms of *fold*:

$$foldl\ f\ v\ xs \quad = \quad fold\ (\lambda x\ g \rightarrow (\lambda a \rightarrow g\ (f\ a\ x)))\ id\ xs\ v$$

In contrast, it is not possible to redefine *fold* in terms of *foldl*, due to the fact that *foldl* is strict in the tail of its list argument but *fold* is not. There are a number of useful 'duality theorems' concerning *fold* and *foldl*, and also some guidelines for deciding which operator is best suited to particular applications (Bird, 1998).

5.2 Ackermann's function

For our final example of the power of *fold*, consider the function *ack* that processes two lists of integers, and is defined using explicit recursion as follows:

$$\begin{array}{lll} ack & :: & [Int] \to ([Int] \to [Int]) \\ ack\ \ [\,]\ \ ys & = & 1 : ys \\ ack\ \ (x : xs)\ \ [\,] & = & ack\ \ xs\ \ [1] \\ ack\ \ (x : xs)\ \ (y : ys) & = & ack\ \ xs\ \ (ack\ \ (x : xs)\ \ ys) \end{array}$$

This is Ackermann's function, converted to operate on lists rather than natural numbers by representing each number n by a list with n arbitrary elements. This function is the classic example of a function that is not primitive recursion in a first-order programming language. However, in a higher-order language such as Haskell, Ackermann's function is indeed primitive recursive (Reynolds, 1985). In this section we show how to calculate the definition *ack* in terms of *fold*.

First of all, by appealing to the universal property of *fold*, the equation $ack = fold\ \ f\ \ v$ is equivalent to the following two equations:

$$\begin{array}{lll} ack\ \ [\,] & = & v \\ ack\ \ (x : xs) & = & f\ \ x\ \ (ack\ \ xs) \end{array}$$

A simple calculation from the first equation gives the definition $v = (1 :)$. From the second equation, proceeding in the normal manner does not result in a definition for the function f, as the reader may wish to verify. However, progress can be made by first using *fold* to redefine the function $ack\ \ (x : xs)$ on the left-hand side of the second equation. By appealing to the universal property, the equation $ack\ \ (x : xs) = fold\ \ g\ \ w$ is equivalent to the following two equations:

$$\begin{array}{lll} ack\ \ (x : xs)\ \ [\,] & = & w \\ ack\ \ (x : xs)\ \ (y : ys) & = & g\ \ y\ \ (ack\ \ (x : xs)\ \ ys) \end{array}$$

The first equation gives $w = ack\ \ xs\ \ [1]$, and from the second:

$$\begin{array}{ll} & ack\ \ (x : xs)\ \ (y : ys) = g\ \ y\ \ (ack\ \ (x : xs)\ \ ys) \\ \Leftrightarrow & \quad \{\ \text{Definition of } ack\ \} \\ & ack\ \ xs\ \ (ack\ \ (x : xs)\ \ ys) = g\ \ y\ \ (ack\ \ (x : xs)\ \ ys) \\ \Leftarrow & \quad \{\ \text{Generalising } (ack\ \ (x : xs)\ \ ys) \text{ to } zs\ \} \\ & ack\ \ xs\ \ zs = g\ \ y\ \ zs \\ \Leftrightarrow & \quad \{\ \text{Functions}\ \} \\ & g = \lambda y \to ack\ \ xs \end{array}$$

That is, using the universal property we have calculated that:

$$ack\ \ (x : xs) \quad = \quad fold\ \ (\lambda y \to ack\ \ xs)\ \ (ack\ \ xs\ \ [1])$$

Using this result, we can now calculate a definition for f:

$$\begin{array}{ll} & ack\ \ (x : xs) = f\ \ x\ \ (ack\ \ xs) \\ \Leftrightarrow & \quad \{\ \text{Result above}\ \} \\ & fold\ \ (\lambda y \to ack\ \ xs)\ \ (ack\ \ xs\ \ [1]) = f\ \ x\ \ (ack\ \ xs) \\ \Leftarrow & \quad \{\ \text{Generalising } (ack\ \ xs) \text{ to } g\ \} \\ & fold\ \ (\lambda y \to g)\ \ (g\ \ [1]) = f\ \ x\ \ g \\ \Leftrightarrow & \quad \{\ \text{Functions}\ \} \\ & f = \lambda x\ g \to fold\ \ (\lambda y \to g)\ \ (g\ \ [1]) \end{array}$$

In summary, using the universal property *twice* we have calculated that:

$$ack \quad = \quad fold\ \ (\lambda x\ g \to fold\ \ (\lambda y \to g)\ \ (g\ \ [1]))\ \ (1 :)$$

6 Other work on recursion operators

In this final section we briefly survey a selection of other work on recursion operators that we did not have space to pursue in this article.

Fold for regular datatypes. The fold operator is not specific to lists, but can be generalised in a uniform way to 'regular' datatypes. Indeed, using ideas from category theory, a single fold operator can be defined that can be used with any regular datatype (Malcolm, 1990*b*; Meijer *et al.*, 1991; Sheard and Fegaras, 1993).

Fold for nested datatypes. The fold operator can also be generalised in a natural way to 'nested' datatypes. However, the resulting operator appears to be too general to be widely useful. Finding solutions to this problem is the subject of current research (Bird and Meertens, 1998; Jones and Blampied, 1998).

Fold for functional datatypes. Generalising the fold operator to datatypes that involve functions gives rise to technical problems, due to the contravariant nature of function types. Using ideas from category theory, a fold operator can be defined that works for such datatypes (Meijer and Hutton, 1995*a*), but the the use of this operator is not well understood, and practical applications are lacking. However, a simpler but less general solution has given rise to some interesting applications concerning cyclic structures (Fegaras and Sheard, 1996).

Monadic fold. In a series of influential articles, Wadler showed how pure functional programs that require imperative features such as state and exceptions can be modelled using monads (Wadler, 1990, 1992*a*, 1992*b*). Building on this work, the notion of a 'monadic fold' combines the use of fold operators to structure the processing of recursive values with the use of monads to structure the use of imperative features (Fokkinga, 1994; Meijer and Jeuring, 1995*b*).

Relational fold. The fold operator can also be generalised in a natural way from functions to relations. This generalisation supports the use of fold as a specification construct, in addition to its use as a programming construct. For example, a relational fold is used in the circuit design calculus Ruby (Jones and Sheeran, 1990; Jones, 1990), the Eindhoven spec calculus (Aarts *et al.*, 1992), and in a recent textbook on the algebra of programming (Bird and de Moor, 1997).

Other recursion operators. The fold operator is not the only useful recursion operator. For example, the dual operator unfold for constructing rather than processing recursive values has been used for specification purposes (Jones, 1990; Bird and de Moor, 1997), to program reactive systems (Kieburtz, 1998), to program operational semantics (Hutton, 1998), and is the subject of current research. Other interesting recursion operators include the so-called paramorphisms (Meertens, 1992), hylomorphisms (Meijer, 1992), and zygomorphisms (Malcolm, 1990*a*).

Automatic program transformation. Writing programs using recursion operators can simplify the process of optimisation during compilation. For example, eliminating the use of intermediate data structures in programs (deforestation) in considerably simplified when programs are written using recursion operators rather than general recursion (Wadler, 1981; Launchbury and Sheard, 1995; Takano and Meijer, 1995). A generic system for transforming programs written using recursion operators is currently under development (de Moor and Sittampalan, 1998).

Polytypic programming. Defining programs that are not specific to particular datatypes has given rise to a new field, called polytypic programming (Backhouse *et al*., 1998). Formally, a polytypic program is one that is parameterised by one or more datatypes. Polytypic programs have already been defined for a number of applications, including pattern matching (Jeuring, 1995), unification (Jansson and Jeuring, 1998), and various optimisation problems (Bird and de Moor, 1997).

Programming languages. A number of experimental programming languages have been developed that focus on the use of recursion operators rather than general recursion. Examples include the algebraic design language ADL (Kieburtz and Lewis, 1994), the categorical programming language Charity (Cockett and Fukushima, 1992), and the polytypic programming language PolyP (Jansson and Jeuring, 1997).

Acknowledgements

I would like to thank Erik Meijer and the members of the Languages and Programming group in Nottingham for many hours of interesting discussions about *fold*. I am also grateful to Roland Backhouse, Mark P. Jones, Philip Wadler, and the anonymous *JFP* referees for their detailed comments on the article, which led to a substantial improvement in both the content and presentation. This work is supported by Engineering and Physical Sciences Research Council (EPSRC) research grant GR/L74491, Structured Recursive Programming.

References

Aarts, C., Backhouse, R., Hoogendijk, P., Voermans, E. and van der Woude, J. (1992) *A relational theory of datatypes*. Available on the World-Wide-Web: `http://www.win.tue.nl/win/cs/wp/papers/papers.html`.

Backhouse, R., Jansson, P., Jeuring, J. and Meertens, L. (1998) Generic programming: An introduction. *Lecture Notes of the 3rd International Summer School on Advanced Functional Programming*.

Backus, J. (1978) Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, **9**.

Barendregt, H. (1984) *The Lambda Calculus – Its syntax and semantics*. North-Holland.

Bird, R. (1989) Constructive functional programming. *Proc. Marktoberdorf International Summer School on Constructive Methods in Computer Science*. Springer-Verlag.

Bird, R. (1998) *Introduction to Functional Programming using Haskell* (*2nd ed*.). Prentice Hall.

Bird, R. and de Moor, O. (1997) *Algebra of Programming*. Prentice Hall.

Bird, R. and Meertens, L. (1998) Nested datatypes. In: Jeuring, J. (ed.), *Proc. 4th International Conference on Mathematics of Program Construction: Lecture Notes in Computer Science 1422*. Springer-Verlag.

Bird, R. and Wadler, P. (1988) *An Introduction to Functional Programming*. Prentice Hall.

Cockett, R. and Fukushima, T. (1992) About Charity. *Yellow Series Report No. 92/480/18*, Department of Computer Science, The University of Calgary.

de Moor, O. and Sittampalan, G. (1998) Generic program transformation. *Lecture Notes of the 3rd International Summer School on Advanced Functional Programming*.

Fegaras, L. and Sheard, T. (1996) Revisiting catemorphisms over datatypes with embedded

functions. *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Fokkinga, M. (1994) Monadic maps and folds for arbitrary datatypes. *Memoranda Informatica 94-28*, University of Twente.

Fokkinga, M., Jeuring, J., Meertens, L. and Meijer, E. (1991) Translating attribute grammars into catamorphisms. *The Squiggolist*, **2**(1).

Hutton, G. (1998) Fold and unfold for program semantics. *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming*.

Iverson, K. E. (1962) *A Programming Language*. Wiley.

Jansson, P. and Jeuring, J. (1997) PolyP – a polytypic programming language extension. *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press.

Jansson, P. and Jeuring, J. (1998) *Polytypic unification. J. Functional Programming* (to appear).

Jeuring, Jo. (1995) Polytypic pattern matching. *Proc. 7th International Conference on Functional Programming and Computer Architecture*. ACM Press.

Jones, G. (1990) Designing circuits by calculation. *Technical Report PRG-TR-10-90*, Oxford University.

Jones, G. and Sheeran, M. (1990) Circuit design in Ruby. In: Staunstrup (ed.), *Formal Methods for VLSI Design*. Elsevier.

Jones, M. P. and Blampied, P. (1998) A pragmatic approach to maps and folds for parameterized datatypes. Submitted.

Kieburtz, R. B. (1998) Reactive functional programming. *Proc. PROCOMET*. Chapman & Hall.

Kieburtz, R. B. and Lewis, J. (1994) *Algebraic Design Language* (*preliminary definition*). Oregon Graduate Institute of Science and Technology.

Kleene, S. C. (1952) *Introduction to Metamathematics*. Van Nostrand Rheinhold.

Launchbury, J. and Sheard, T. (1995) Warm fusion: Deriving build-catas from recursive definitions. *Proc. 7th International Conference on Functional Programming and Computer Architecture*. ACM Press.

Malcolm, G. (1990*a*) *Algebraic data types and program transformation*. PhD thesis, Groningen University.

Malcolm, G. (1990*b*) Algebraic data types and program transformation. *Science of Computer Programming*, **14**(2–3), 255–280.

Meertens, L. (1983) Algorithmics: Towards programming as a mathematical activity. *Proc. CWI Symposium*.

Meertens, L. (1992) Paramorphisms. *Formal Aspects of Computing*, **4**(5), 413–425.

Meijer, E. (1992) *Calculating compilers*. PhD thesis, Nijmegen University.

Meijer, E. and Hutton, G. (1995*a*) Bananas in space: Extending fold and unfold to exponential types. *Proc. 7th International Conference on Functional Programming and Computer Architecture*. ACM Press.

Meijer, E. and Jeuring, J. (1995*b*) Merging monads and folds for functional programming. In: Jeuring, J. and Meijer, E. (eds.), *Advanced Functional Programming: Lecture Notes in Computer Science 925*. Springer-Verlag.

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.), *Proc. Conference on Functional Programming and Computer Architecture: Lecture Notes in Computer Science 523*. Springer-Verlag.

Peterson, J. *et al.* (1997) *The Haskell language report, version 1.4*. Available on the World-Wide-Web: `http://www.haskell.org`.

Reynolds, J. C. (1985) Three approaches to type structure. *Proc. International Joint Conference on Theory and Practice of Software Development: Lecture Notes in Computer Science 185*. Springer-Verlag.

Sheard, T. and Fegaras, L. (1993) A fold for all seasons. *Proc. ACM Conference on Functional Programming and Computer Architecture*. Springer-Verlag.

Swierstra, S. D., Alcocer, P. R. A. and Saraiva, J. (1998) Designing and implementing combinator languages. *Lecture Notes of the 3rd International Summer School on Advanced Functional Programming*.

Takano, A. and Meijer, E. (1995) Shortcut deforestation in calculational form. *Proc. 7th International Conference on Functional Programming and Computer Architecture*. ACM Press.

Wadler, P. (1981) Applicative style programming, program transformation, and list operators. *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*.

Wadler, P. (1990) Comprehending monads. *Proc. ACM Conference on Lisp and Functional Programming*.

Wadler, P. (1992*a*). The essence of functional programming. *Proc. Principles of Programming Languages*.

Wadler, P. (1992*b*) Monads for functional programming. In: Broy, M. (ed.), *Proc. Marktoberdorf Summer School on Program Design Calculi*. Springer-Verlag.

Notes on rewriting

Robin Cockett
Department of Computer Science
University of Calgary

October 23, 2008

1 Introduction

Rewriting is a fundamental technique both in algebra and in programming languages. These notes are aimed at giving a brief introduction to rewriting theory on algebraic systems.

The main topics covered are:

- Termination using well-founded orderings;
- Local confluence and confluence: Newman's lemma that termination and local confluence implies confluence;
- Equality and confluence: the uniqueness of normal forms;
- The confluence of left-linear orthogonal rewrite systems;
- Standardization: normal forms for rewriting sequences;
- A brief discussion of combinatory logic and functional completeness.

2 Rewriting on algebraic systems

An algebraic system is determined by:

- A set of function symbols Ω each of which has an associated arity:

$$\mathsf{arity} : \Omega \longrightarrow \mathbb{N}; f \mapsto \mathsf{arity}(f)$$

- A set of terms, $T_\Omega(X)$ built inductively from a set of variables X as follows:

$$\frac{x \in X}{x \in T_\Omega(X)} \text{ variable}$$

$$\frac{t_1, ..., t_n \in T_\Omega(X) \quad \mathsf{arity}(f) = n}{f(t_1, .., t_n) \in T_\Omega(X)} \text{ function application}$$

A rewriting system on an algebraic system is generated by a set of **primitive rewritings**

$$R \subseteq T_\Omega(X) \times T_\Omega(X)$$

where we require, when $(t_1, t_2) \in R$, that $FV(t_1) \supseteq FV(t_2)$, that is the (free) variables of t_1 contain those of t_2. The primitive rewritings generate a (larger) relation, which we shall write $t_1 \underset{R}{\longrightarrow} t_2$, on the terms $T_\Omega(X)$ as follows:

$$\frac{(t_1, t_2) \in R \quad \sigma \text{ a substitution}}{t_1[\sigma] \underset{R}{\longrightarrow} t_2[\sigma]} \text{ substitution}$$

$$\frac{s_1, .., \widehat{s_i}, .., s_n \in T_\Omega(X) \quad t \underset{R}{\longrightarrow} t' \quad \mathsf{arity}(f) = n}{f(s_1, ..., t, ..., s_n) \underset{R}{\longrightarrow} f(s_1, ..., t', ..., s_n)} \text{ application}$$

We shall denote by $t \underset{R}{\longrightarrow}^+ t'$ the transitive closure of the above relation and by $t \underset{R}{\longrightarrow}^* t'$ the transitive, reflexive closure of the relation. In the sequel we shall drop the R as it will be understood from the context.

2.1 Examples of rewriting systems

2.1.1 Monoids

A monoid is an algebraic system with a binary multiplication, $\_ \cdot \_$, and a unit, sfe, which is a constant (this means the arity is 0):

$$\mathsf{arity} : \Omega = \{\cdot, \mathsf{e}\} \longrightarrow \mathbb{N}; \quad \begin{array}{lll} \cdot & \mapsto & 2 \\ & \mapsto & 0 \end{array}$$

a monoid must satisfy equations which here we orient to create a rewriting system:

$$\begin{aligned} (x \cdot y) \cdot z &\longrightarrow x \cdot (y \cdot z) && (1) \\ \mathsf{e} \cdot x &\longrightarrow x && (2) \\ x \cdot \mathsf{e} &\longrightarrow x && (3) \end{aligned}$$

Examples of monoids include: the natural numbers under addition (unit is 0), scalars under multiplication (unit is 1), matrices under multiplication (unit is the diagonal matrix).

2.1.2 Groups

A group is a monoid which adds one more operation called inverse which is a unary operation $(\_)^{-1}$ written traditionally as a superscript. We than have:

$$\mathsf{arity} : \Omega = \{\cdot, \mathsf{e}, (\_)^{-1}\} \longrightarrow \mathbb{N}; \quad \begin{array}{lll} \cdot & \mapsto & 2 \\ \mathsf{e} & \mapsto & 0 \\ (\_)^{-1} & \mapsto & 1 \end{array}$$

a group must satisfy equations, which includes those of a monoid which here we orient to create a rewriting system:

$$\begin{array}{rcl}(x \cdot y) \cdot z & \longrightarrow & x \cdot (y \cdot z) \\ \mathsf{e} \cdot x & \longrightarrow & x \\ x \cdot \mathsf{e} & \longrightarrow & x \\ x \cdot x^{-1} & \longrightarrow & \mathsf{e} \\ x^{-1} \cdot x & \longrightarrow & \mathsf{e}\end{array}$$

Neither the natural numbers under addition nor the scalars under multiplication form groups as they lack inverses (the former because of the lack of negative numbers and the latter because 0 does not have an inverse). However, the non-zero scalars (of a field) from a group and the integers under addition form a group. Also the invertible matrices under multiplication form a group.

2.1.3 Combinatory algebra

A combinatory algebra consists of a binary operation, called application, written as $\_ \bullet \_$ and two constants k and s:

$$\mathsf{arity} : \Omega = \{\bullet, \mathsf{k}, \mathsf{s}\} \longrightarrow \mathbb{N}; \begin{array}{rcl}\bullet & \mapsto & 2 \\ \mathsf{k} & \mapsto & 0 \\ \mathsf{s} & \mapsto & 0\end{array}$$

These must satisfy two equations which we orient:

$$\begin{array}{rcl}(\mathsf{k} \bullet x) \bullet y & \longrightarrow & x \\ ((\mathsf{s} \bullet x) \bullet y) \bullet z & \longrightarrow & (x \bullet z) \bullet (y \bullet z)\end{array}$$

The main example of a combinator algebra which we have is the closed terms of the λ-calculus where application is application and

$$\mathsf{k} = \lambda xy.x \quad \text{and} \quad \mathsf{s} = \lambda xyz.xz(yz).$$

Clearly β-reduction corresponds to the rewritings above ...

2.1.4 BCK-algebra

A variant on a combinatory algebra consists is a BCK-algebra. Again there is a binary operation, called application, written as $\_ \bullet \_$ but this time three constants k and b, c and k:

$$\mathsf{arity} : \Omega = \{\bullet, \mathsf{b}, \mathsf{c}, \mathsf{k}\} \longrightarrow \mathbb{N}; \begin{array}{rcl}\bullet & \mapsto & 2 \\ \mathsf{b} & \mapsto & 0 \\ \mathsf{c} & \mapsto & 0 \\ \mathsf{k} & \mapsto & 0\end{array}$$

These must satisfy the equations which we orient:

$$\begin{array}{rcl}((\mathsf{b} \bullet x) \bullet y) \bullet z & \longrightarrow & (x \bullet z) \bullet y \\ ((\mathsf{c} \bullet x) \bullet y) \bullet z & \longrightarrow & x \bullet (y \bullet z) \\ (\mathsf{k} \bullet x) \bullet y & \longrightarrow & x\end{array}$$

BCK-algebras have a particularly simple rewriting theory.

3 Termination

A rewriting system is **terminating** if the rewriting relation is **well-founded** in the sense that every non-empty set has a least element (this is a normal form relative to that set). This means equivalently that there are no infinite chain of rewrites

$$t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow ...$$

as such would not have a minimal element. So, for example, β-reduction on the λ-calculus is not terminating.

Lemma 3.1 *If a relation is well-founded its transitive closure (here $t \xrightarrow{+} t'$) is also well-founded.*

PROOF: Suppose S is a subset we must find a minimal element in the set ... However, we do not change this minimal element if we upclose the set

$$\Uparrow S = \{t \xrightarrow{+} t' | t' \in S\}.$$

This set has a minimal element with respect to the original $\longrightarrow$ as it is well-founded which must also therefore be minimal with respect to $\xrightarrow{+}$. □

In the case that the number of possible rewrites leaving any term is always finite (this means the term contains a finite number of redexes) then there is a bound on the number of rewrites which a chain leaving that term can have. Having a bound is, logically, a strictly stronger property than being well-founded. However, in many rewrite systems finding an explicit bound (that is a natural number) on the number of rewrites is quite straightforward.

Lemma 3.2 *A rewriting system $\mathcal{R}$ is terminating if and only if there is a well-founded set $(W, <)$ and a map $\alpha : T_\Omega(X) \longrightarrow W$ such that:*

- *$\alpha(r_i) > \alpha(c_i)$ for each $r_i \longrightarrow c_i \in \mathcal{R}$;*
- *If $\alpha(t) > \alpha(t')$ then $\alpha(t[\sigma]) > \alpha(t'[\sigma])$;*
- *If $\alpha(t_i) > \alpha(t_i')$ then $\alpha(f(t_1, ..., t_i, ..., t_n)) > \alpha(f(t_1, ..., t_i', ..., t_n))$.*

PROOF: If the rewriting system is terminating then $t \xrightarrow{=} t'$ is well founded so we take α to be the identity map. Conversely given such an α suppose $\emptyset \neq S \subseteq T_\Omega(X)$ then the set $\{\alpha(s) | s \in S\} \subseteq W$ is non-empty and therefore has a minimal element $\alpha(t_0)$. However if $t_0 \longrightarrow t_0'$ is a reduction step which has $t_0' \in S$ then $\alpha(t_0') < \alpha(t_0)$ contradicting the minimality of $\alpha(t_0)$. Thus, t_0 is a minimal element in S with respect to the reduction order showing this order is well-founded. □

Thus, to demonstrate that a rewrite system terminates it suffices to exhibit a map α satisfying the conditions of this lemma.

3.1 Examples of termination arguments

3.1.1 Rewriting for monoids

The rewrite system for monoids is terminating:

Consider the "cost" function $b : T_\Omega(X) \longrightarrow \mathbb{N}$ given by:

$$\begin{aligned} \alpha(x) &= 1 \\ \alpha(\mathsf{e}) &= 1 \\ \alpha(t \cdot t') &= 2\alpha(t) + \alpha(t') \end{aligned}$$

I claim this gives a bound for the rewriting. To see this it suffices to show that if $t \longrightarrow t'$ then $b(t) < b(t)$ so that each rewriting strictly decreases the cost. As $\mathbb{N}$ is well-founded this ensures the rewriting is well founded.

Note that (a) each terms has cost at least 1 (b) decreasing the cost of a subterm will result in at least that amount of cost decrease for the whole term. This means that it suffices to check that the rewrites applied at the root of the term always strictly decrease the cost. However, this is a simple calculation:

$$\begin{aligned} \alpha((t_1 \cdot t_2) \cdot t_3) &= 4\alpha(t_1) + 2\alpha(t_2) + \alpha(t_3) > 2\alpha(t_1) + 2\alpha(t_2) + \alpha(t_3) = \alpha(t_1 \cdot (t_2 \cdot t_3) \\ \alpha(\mathsf{e} \cdot t) &= 2 + \alpha(t) > \alpha(t) \\ \alpha(t \cdot \mathsf{e}) &= 2\alpha(t) + 1 > \alpha(t). \end{aligned}$$

3.2 Rewriting for BCK-algebras

Consider the cost function which simply counts occurrences of b, c, and k:

$$\begin{aligned} \alpha(x) &= 1 \quad x \text{ is a variable} \\ \alpha(\mathsf{b}) = \alpha(\mathsf{c}) = \alpha(\mathsf{k}) &= = 1 \\ \alpha(t_1 \bullet t_2) &= \alpha(t_1) + \alpha(t_2) \end{aligned}$$

It is easy to see that this cost function descreases across all the rewrites for BCK-algebras.

3.3 Well-founded strict partial orders

In constructing more sophisticated termination arguments it is often useful to use more complex well-founded orders. The gandfather of all well-founded orders is the (strict) order on the natural numbers (this is also a stricttotal order). There are a number of important ways of constructing well-founded strict partial orders from other well-founded strict partial orders:

3.3.1 Lexicographical orderings on strings

If P is well-founded then P^* with the lexicographical ordering. The lexicographical ordering is given by [] is minimal and $x : xs < y : ys$ if either $x < y$ or if $x = y$ and $xs < ys$.

Lemma 3.3 *The lexicographical relation on P^* derived from a well-founder relation on P is itself well-founded.*

PROOF: This strict inequality is well-founded as given any $X \subset P^*$ if the empty list is in there is automatically a minimal element otherwise choose a minimal element $x_0 \in P$ in the first entry and consider the tails $\mathsf{Tail}_{x_0}(X) = \{xs | x_0 : xs \in X\}$, if there is a minimal element in this set then there is a minimal element in X: in fact, if we continue this process this amounts to asking that one of the tails sets contains the empty list. Suppose for contradiction that this never happens then we would have an element $p_0 \in P^*$ which is in all the sets

$$X \supseteq x_0 : \mathsf{Tail}_{x_0}(X) \supseteq x_0 : x_1 : \mathsf{Tail}_{x_0}(\mathsf{Tail}_{x_0}(X)) \supseteq x_0 : x_1 : x_2 : \mathsf{Tail}_{x_2}(\mathsf{Tail}_{x_1}(\mathsf{Tail}_{x_0}(X))) \supseteq ...$$

but p_0 has a finite length so one of these sets must contain the empty list. □

3.3.2 Lexicographical ordering of Rose trees

We consider trees with entries at the nodes and a finite list of subtrees (that is `data Rose a = (;) a [Rose a]`) these are called Rose trees. Rose trees of a well-founded poset form a well-founded set. We set $x; xs < y; ys$ if $x < y$ or if $x = y$ and $xs < ys$ (where we use the lexicographical ordering of the subtrees).

Lemma 3.4 *The lexiographical relation on Rose trees,* $\mathsf{Rose}(P)$ *based on a well-founded relation on* P *is itself well-founded.*

PROOF: The argument that this strict ordering is well-founded is almost identical to the argument we have just done: in any set of rose trees there are trees with a minimal element element at the root we may than consider the set of lists of arguments $\mathsf{Tails}_{x_0}(X)$ if this contains a minimal element we are done. At any rate either this contains the empty list (so we are done) or we may choose a lexicographically least list of first elements. This allows us to grow a prefix of a Rose tree agreeing with elements in X which are minimal so far. The same argument as before shows that this growing process must close off with empty lists of argments as a given Rose tree which is in all these sets is finite. □

3.3.3 Bags ordering

A bag is an unordered list: this means repetitions are allowed: $\{\!\{x, y, x\}\!\} = \{\!\{y, x, x\}\!\} \neq \{\!\{x, y\}\!\}$ but the order in which the elements occur does not matter. The set $\mathsf{Bag}(P)$ of bags of elements of a set P with a well founded relation can be endowed with a well-founded realtion. One bag is less than another in case:

- The empty bag $\{\!\{\}\!\}$ is minimal.
- $\{\!\{x\}\!\} \sqcup b_1 < \{\!\{x\}\!\} \sqcup b_2$ if $b_1 < b_2$;
- $\{\!\{x_1, .., x_n\}\!\} \sqcup b < \{\!\{y\}\!\} \sqcup b$ whenever each $x_i < y$.

Thus we may determine whether one bag is (strictly) less than the other by first removing the elements in common and then for each element in the bigger bag removing all elements in the smaller bag which are strictly less than it. The smaller bag will be emptied by this process.

Intuitively to construct an element strictly smaller than a given b we are allowed to pick an element from the bag and either remove it or replace it with a bag of elements each of which is individually strictly smaller than the element removed. It is not so obvious that this process must always end as you may replace the element with a very large bag of smaller elements!

Proposition 3.5 *If P is a well founded set then the induced relation on* $\mathsf{Bag}(P)$ *given above is well founded.*

PROOF: To prove this is well-founded is not so easy (Dershowitz and Manna). Here is a sketch: suppose there is an infinite descending chain of bags $b_0 < b_1 < b_2 < ...$ then we may build a forest with roots $x \in b_0$ the children of x are the bag $\{x_1, .., x_n\}$ of elements which eventually replace x that is $b_i < b_{i+1}$ (for some i) introduces this strict replacement of x (if this never happens then x is a leaf). Every step in the sequence must do one (or more) replacements of this nature. However, this tree is finitely branching and has all its paths of finite length so itself is finite (Konigs lemma). This means the sequence itself must be finite. □

3.3.4 Bag trees:

A bag tree is a rose tree in which the order of the children does not matter (that is morally `data BagTree a = (.) a (Bag (Bagtree a))`). If P is a well-founded set then $\mathsf{BagTree}(P)$ is also well-founded. The order is given inductively using the bag ordering: $x.\{\!\{t_1, .., t_n\}\!\} \leq y.\{\!\{s_1, ..., s_m\}\!\}$ if $x < y$ or when $x = y$ when $\{\!\{t_1, .., t_n\}\!\} < \{\!\{s_1, ..., s_m\}\!\}$.

Now, in fact, an unordered tree may equally be represented by its bag of paths. The ordering I have just described is just the bag ordering on the lexicographical ordering on the paths! Therefore it is certainly well-founded.

3.3.5 Recursive path ordering

There is a more sophisticated family of well-founded ordering on unordered trees which are called *recursive path orderings* they may be given (following Klop) by the transitive closure of a rewrite system between unordered trees (where one adds a unary operation $(\_)^*$) restricted to the unordered trees (so you must eventually remove the added symbol!).

$$\begin{aligned}
n.b &\longrightarrow n^*.b \\
n^*.b &\longrightarrow m.\{\!\{n^*.b, ..., n^*.b\}\!\} \qquad m < n \\
n^*.\{\!\{m.b\}\!\} \sqcup b' &\longrightarrow n.\{\!\{m^*.b, ..., m^*b\}\!\} \sqcup b' \quad \text{(where any number of copies is permitted)} \\
n^*.\{\!\{m.b\}\!\} \sqcup b' &\longrightarrow m.b
\end{aligned}$$

See Klop for the proof.

Here is an exercise to show that the rewriting system determined on terms by $x \cdot (x + y) \longrightarrow (x \cdot x) + (x \cdot y)$ and $(x + y) + z \longrightarrow x + (y + z)$ is terminating?

An illustration of recursive path ordering (Klop) is for the rewrite system

$$\begin{aligned}
\neg\neg y &\longrightarrow y \\
\neg(x \vee y) &\longrightarrow (\neg x) \wedge (\neg y) \\
\neg(x \wedge y) &\longrightarrow (\neg x) \vee (\neg y) \\
x \wedge (y \vee z) &\longrightarrow (x \wedge y) \vee (x \wedge z) \\
(y \vee z) \wedge x &\longrightarrow (y \wedge x) \vee (z \wedge x)
\end{aligned}$$

We map terms to unordered trees on the natural numbers:

$$V(x \vee y) = 1.\{\!\{V(x), V(y)\}\!\} \quad V(x \wedge y) = 2.\{\!\{V(x), V(y)\}\!\} \quad V(\neg x) = 3.\{\!\{V(x)\}\!\}$$

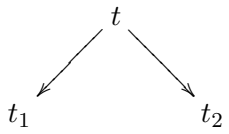
then check that each rule is a reduction here are some of the verifications (exercise complete the set):

$$\begin{aligned}
\neg\neg x &= 3.\{\!|3.\{\!|V(x)|\!\}|\!\} \longrightarrow 3^*.\{\!|3.\{\!|V(x)|\!\}|\!\} \longrightarrow 3.\{\!|V(x)|\!\} \longrightarrow 3^*.\{\!|V(x)|\!\} \longrightarrow V(x) \\
\neg(x \vee y) &= 3.\{\!|1.\{\!|V(x), V(y)|\!\}|\!\} \longrightarrow 3^*.\{\!|1.\{\!|V(x), V(y)|\!\}|\!\} \\
&\longrightarrow 1.\{\!|3^*.\{\!|1.\{\!|V(x), V(y)|\!\}|\!\}, 3^*.\{\!|1.\{\!|V(x), V(y)|\!\}|\!\}|\!\} \\
&\longrightarrow 2.\{\!|3.\{\!|1^*.\{\!|V(x), V(y)|\!\}|\!\}, 3.\{\!|1^*.\{\!|V(x), V(y)|\!\}|\!\}|\!\} \\
&\longrightarrow 2.\{\!|3.\{\!|V(x)|\!\}, 3.\{\!|V(y)|\!\}|\!\} = V((\neg x) \wedge (\neg y))
\end{aligned}$$

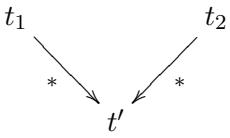
It is worth noting that recursive path ordering cannot, in general, handle associative laws. One way to recover this ability is to add subscripts to the operation indicating its left depth (for example) ...

4 Confluence

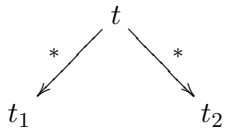
We shall say that a rewriting system is **locally confluent** if every one-step divergence



has a (possibly) multi-step convergence:

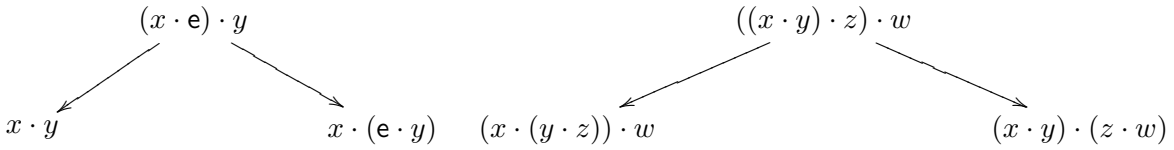


We shall say that a rewriting is **confluent** if every multi-step divergence

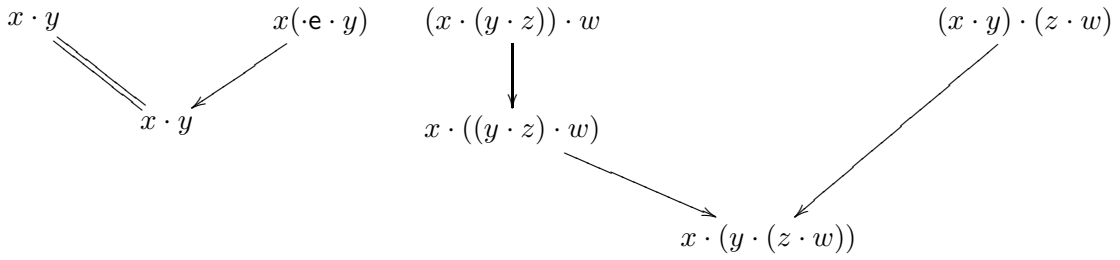


has a convergence as above. Some examples of divergences are:

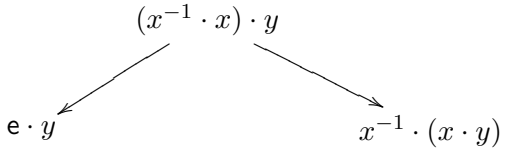
1. Two one-step divergences from the example of monoids are:



For each there are, in this case, corresponding convergences. Note that the first uses the reflexivity non-trivially while the second requires more than on rewriting step on one of the legs:



2. A one step divergence from the example of groups is:



   Notice that this divergence has no corresponding convergence with respect to the rewrite system given so that this rewrite system is not confluent.

With respect to a rewriting relation a term t is said to be in **normal form** if it is minimal in the sense that there is no rewriting $t \longrightarrow t'$.

When rewrite systems become more sophisticated then the fundamental use of termination, in the sense of the rewrite relation being well-founded, becomes more essential. To show termination of simple rewrite systems, often, a cost function into the natural numbers, as above, will do the trick. In this case a standard induction will do the trick. However, assuming that the one-step rewriting is just well-founded forces us to do well-founded inductions. This uses the following principal:

Principle of well-founded induction: Given a well-founded relation (here the one-step rewriting or its transitive closure) we have the following inference for any property:

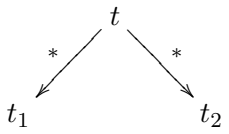
$$\frac{\forall t.(\forall t'.t > t' \Rightarrow P(t')) \Rightarrow P(t)}{\forall t.P(t)} \text{ wf-ind}$$

Notice that this means that the property must be true, in particular, for all t which are minimal (i.e. are in normal form) as these have no elements below them.

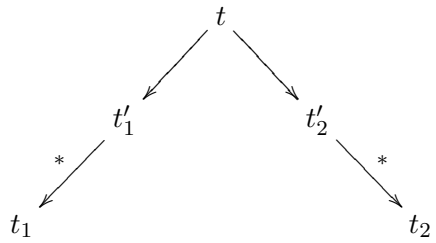
Lemma 4.1 (Newman) *Any terminating rewriting system which is locally confluent is confluent.*

PROOF: The property P we wish to prove is "every divergence leaving t has a convergence. This is therefore certainly true of any elements in normal form. Suppose t is such that whenever t' has $t \xrightarrow{+} t'$ then t' has this property then we argue as follows to show that t has the property:

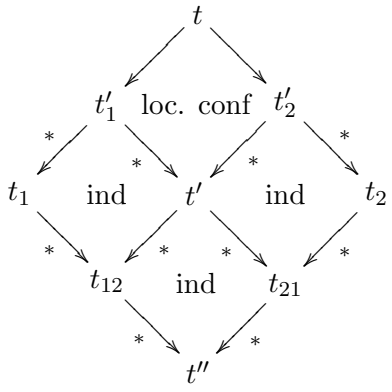
Consider a divergence



if either of the arms is trivial (i.e involves no steps) then there is an obvious convergence. So we may assume that each arm has at least one step. So the divergence is now of the form:



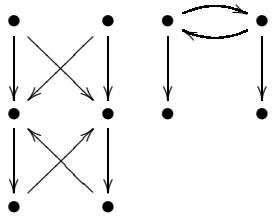
However, everything strictly below t satisfies the induction hypothesis so that:



□

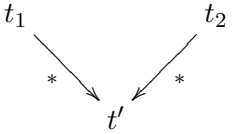
Notice that if a system is confluent then each t has at most one normal form associated with it (there may be none: remember the λ-calculus). This is because suppose that t has two distinct normal forms s_1 and s_2 with $t \xrightarrow{*} s_1$ and $t \xrightarrow{*} s_2$ then this constitutes a divergence. However the convergence forces $s_1 = s_2$. So in a confluent system normal forms are unique.

Here is a rewriting system (between constants in an algebraic theory), in which local confluence holds yet confluence does not hold, in which there are objects with no normal form and with two normal forms (the example follows Huet):



Given a rewriting system if we treat the rewrites as equalities (of an algebraic system) this gives the following observation which explains the preoccupation with confluence:

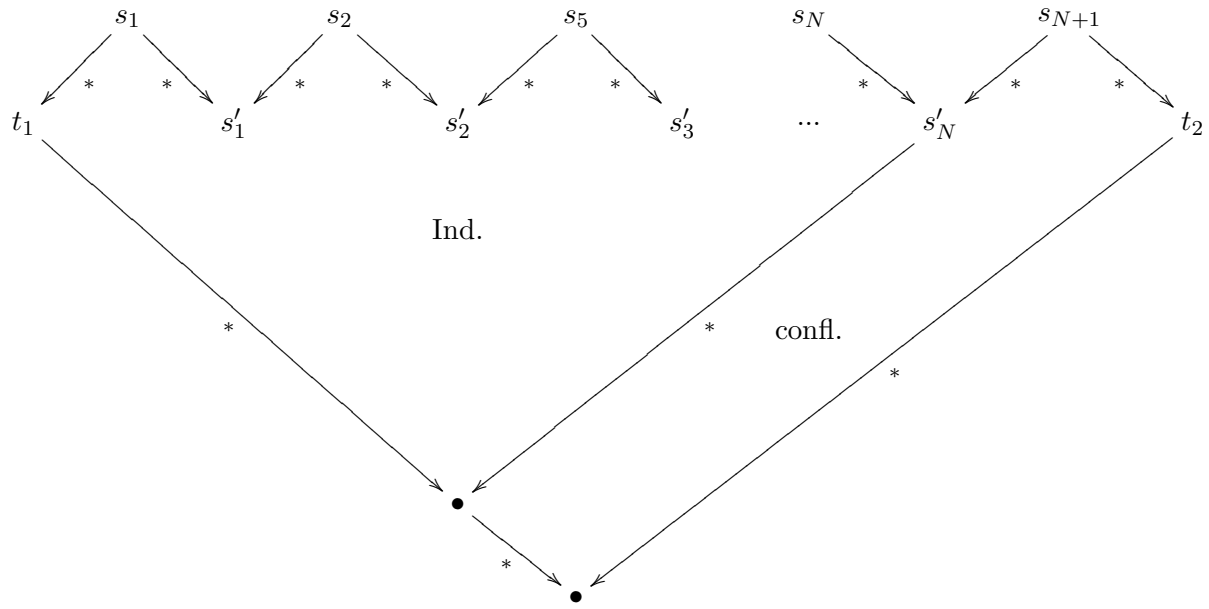
Lemma 4.2 *In a confluent rewriting system $t_1 = t_2$ if and only if there is a convergence*



PROOF: t_1 equals t_2 if and only if there is a zigzag of rewritings:

$$t_1 \xleftarrow{*} s_1 \xrightarrow{*} s_1' \xleftarrow{*} s_2 \xrightarrow{*} s_2' \xleftarrow{*} \ldots s_n \xrightarrow{*} t_2$$

We argue by the number of zigs (in this case n). If n is 0 the result is immediate. It is nice, though not necessary, to observe that when $n = 1$ then it is given by the confluence. We suppose the result true for N and consider a zigzag with $N + 1$ zigs.



Using the induction hypothesis and confluence shows that the result holds for N. □

Corollary 4.3 *In a confluent terminating rewriting system*

(i) *Every term has a normal form;*

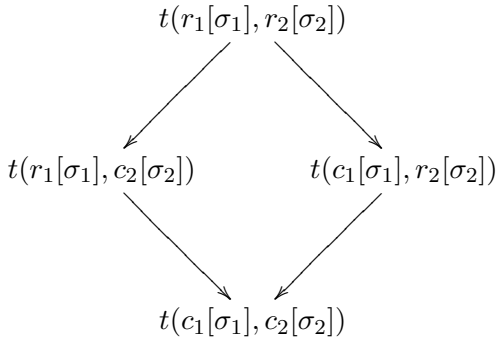
(ii) *Two terms are equal if and only if they have the same normal form.*

5 Critical pairs

In an algebraic system there are (usually) infinitely many terms so that it is not practical to check every divergence for a convergence. It is therefore important to understand which (one step) divergences one actually has to check. To understand this it is useful to understand what a **redex** is: it is the part of the term that a rewriting rule removes in order to replace it with the **contractum**. A redex indicates where in the term a primitive rewriting could be applied and the part of the subterm standing at that place which will be affected by the rewriting.

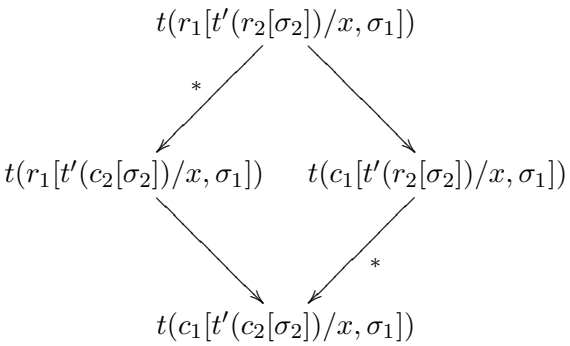
We say that two rewritings $t \longrightarrow t_1$ and $t \longrightarrow t_2$ are **independent** in case the redexes of the primitive rewritings they embody are disjoint. This can happen in two different ways: firstly the redexes can be in parallel parts of the term (i.e. they are below different arguments of some function

symbol). We may write this situation as $t(r_1[\sigma_1], r_2[\sigma_2])$ where $r_1 \longrightarrow c_1$ and $r_2 \longrightarrow c_2$ are the two rewritings. In this situation it is clear that:



so that local confluence is assured.

The other possibility is that one redex is below the other which we represent as $t(r_1[t'(r_2[\sigma_2])/x, \sigma_1])$. There is immediately a little subtly concerning this situation which we should be clear about: in the redex r_1 the variable x may occur more an once. Thus r_2 may occur in multiple (parallel) places and it is only one of these redexes which is the actual second redex we had in mind when we started. However, we do have the following:



and from this we can obtain our local confluence by replacing the downward arrow by a sequence of rewrites the first of which is the chosen rewrite.

We have now shown that:

Lemma 5.1 *In any rewrite system over an algebraic system, rewrites with independent redexes are locally confluent.*

The importance of this is that in order to establish local confluence it suffices to check the cases when the redexes overlap. However, we can go further: clearly all the rewriting takes place on the term below where the first redex starts so we need only consider the subterm at which that redex starts. Finally, notice that the terms which are *just* the overlapping redexes themselves will exhibit the divergence (in fact, in a minimal way). Furthermore, when this minimal divergence has a convergence any substitution of it will also have a convergence. Thus, by solving such a divergence, which we certainly must do anyway, we will have solved all the cases where this pattern of overlapping redexes occur. Thus, it suffices to check the divergences arising from these overlapping redexes for convergence. These minimal divergences are called **critical divergences**

(the two feet are called a **critical pair**). When the rewriting system is terminating we may rewrite the two feet of a critical divergence into a common normal form.

To form critical pairs involves identifying a subterm in a redex (which can be the whole term) at which the overlap will happen. Thus given a redex r_1 one splits it into $r_1 = r_1'(s)$, where s is not a variable, and next one finds a most general substitution, that is a the most general unifier, σ, such that $r_2[\sigma] = s[\sigma]$, where r_2 and s are assumed to have no variables in common. The divergence:

$$\begin{array}{ccc} & r_1[\sigma] = r_1'(s)[\sigma] & \\ \swarrow & & \searrow \\ c_1[\sigma] & & r_1'(c_2)[\sigma] \end{array}$$

is then a critical pair. Notice that this whole construction is more delicate as r_1 can have repeated variables, as in the rewrite $x \cdot x^{-1} \longrightarrow sfe$ above for example, so that simply overlapping r_1 and r_2 in the obvious manner may destroy the redex for r_2 as it requires that two subterms are equal. Using unification maintains this and delivers the minimal way to do this. Thus, the critical divergence from $(x \cdot y) \cdot z$ and $x \cdot x^{-1} \longrightarrow \mathsf{e}$ is

$$x \cdot (y \cdot (x \cdot y)^{-1}) \longleftarrow (x \cot y) \cdot (x \cdot y)^{-1} \longrightarrow \mathsf{e}.$$

Proposition 5.2 *A rewrite system over an algebraic system is locally confluent if and only if all critical divergences can be converged.*

This reduces testing a (finitely presented) system for local conflence to a finite number of tests. Thus it is an easily decidable question.

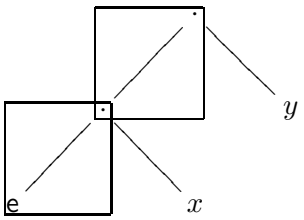
5.1 Examples of determining confluence

5.1.1 Confluence for monoids

Consider the rewriting system for monoids:

At this stage we know the rewriting system is terminating so it remains to look at the *critical pairs* that is minimal terms in which there are overlapping redexes. It is useful here to visualize the terms as trees so that one can see what is going on. There are five critical pairs:

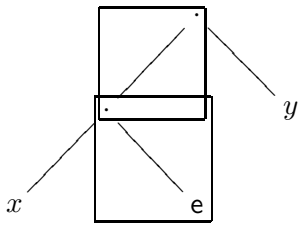
1. Rules (2) and (1) have a critical pair arising from the following overlapping redexes:



This critical pair is resolved as follows:

$$\begin{array}{ccc} (\mathsf{e} \cdot x) \cdot y & \xrightarrow{(1)} & \mathsf{e} \cdot (x \cdot y) \\ {\scriptstyle (2)}\downarrow & & \downarrow{\scriptstyle (2)} \\ x \cdot y & = & x \cdot y \end{array}$$

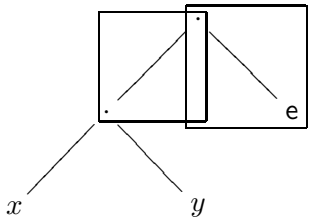
2. Rules (3) and (1) have a first critical pair arising from the following overlapping redexes:



This critical pair is resolved as follows:

$$\begin{array}{ccc} (x \cdot \mathsf{e}) \cdot y & \xrightarrow{(1)} & x \cdot (\mathsf{e} \cdot y) \\ {\scriptstyle (3)}\downarrow & & \downarrow{\scriptstyle (2)} \\ x \cdot y & = & x \cdot y \end{array}$$

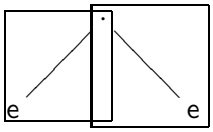
3. Rules (1) and (3) have a critical pair arising from the following overlapping redexes:



This critical pair is resolved as follows:

$$\begin{array}{ccc} (x \cdot y) \cdot \mathsf{e} & \xrightarrow{(1)} & x \cdot (y \cdot \mathsf{e}) \\ {\scriptstyle (3)}\downarrow & & \downarrow{\scriptstyle (3)} \\ x \cdot y & = & x \cdot y \end{array}$$

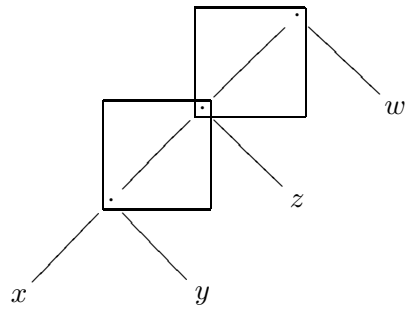
4. Rules (3) and (2) have a critical pair arising from the following overlapping redexes:



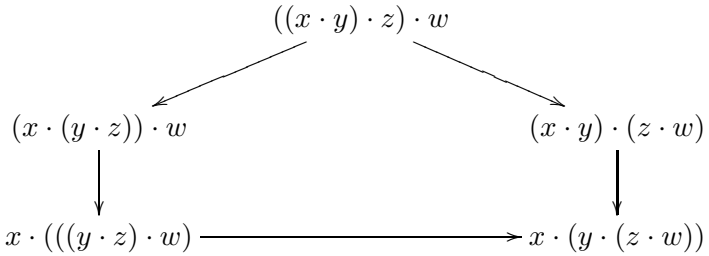
This critical pair is (trivially) resolved as follows:

$$\begin{array}{ccc} \mathsf{e} \cdot \mathsf{e} & \xrightarrow{(3)} & \mathsf{e} \\ {\scriptstyle (2)}\downarrow & & \| \\ \mathsf{e} & = & \mathsf{e} \end{array}$$

5. Rules (1) and (1) have a critical pair arising from the following overlapping redexes:



This critical pair is resolved as follows (this is actually an instance of something very famous sometimes called "the MacLane pentagon"):

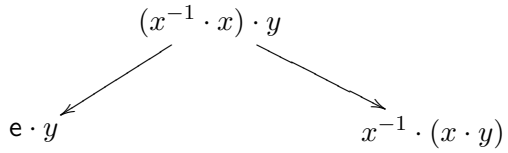


5.1.2 The rewriting system for BCK-algebras is confluent

We have seen that the rewriting for BCK-algebras is terminating. An inspection of the rewrites quickly shows that there are *no* critical pairs for this rewrite system. In fact as we shall shortly see it is an example of an orthogonal rewrite system (which are always confluent). However, even at this stage we can conclude that this rewrite system is confluent and, in fact, rewriting will produce a normal form in linear time.

5.1.3 The rewriting for groups is not confluent

The divergence, mentioned earlier, for the rewriting system for groups:



is a critical divergence which has no convergence. Thus this rewriting system as it stands is not confluent.

5.2 Knuth Bendix completion

A natural further step is to try to complete a rewriting system to make it confluent. This is the Knuth-Bendix completion procedure: the basic idea is as follows: suppose one starts with a

well-founded rewrite system on the terms which agrees with given rewrites and one discovers that a critical divergence cannot be resolved then one adds a rewrite to the system across the pair oriented in the direction of the well-order. This results in an expanded system of rewrites, next one removes any rewrites which are now implied. One then keeps going like this until there are no unresolved critical pairs.

Of course there is no guarantee that this procedure will terminate. However, it gives a very effective way of searching for a confluent rewrite system which often does terminate.

Consider the example of groups. We know the rewriting system as it stands is not confluent as certain critical pairs cannot be resolved:

$$\begin{aligned} \mathsf{e}\cdot y \leftarrow \quad & (x^{-1}\cdot x)\cdot y \quad && \longrightarrow x^{-1}\cdot(x\cdot y) \\ \mathsf{e}\cdot y \leftarrow \quad & (x\cdot x^{-1})\cdot y \quad && \longrightarrow x\cdot(x^{-1}\cdot y) \\ \mathsf{e} \leftarrow \quad & \mathsf{e}\cdot\mathsf{e}^{-1} \quad && \longrightarrow \mathsf{e}^{-1} \\ \mathsf{e}^{-1} \leftarrow \quad & \mathsf{e}^{-1}\cdot\mathsf{e} \quad && \longrightarrow \mathsf{e} \end{aligned}$$

These cause us to add to the system the oriented equations:

$$\begin{aligned} x^{-1}\cdot(x\cdot y) &\longrightarrow y \\ x\cdot(x^{-1}\cdot y) &\longrightarrow y \\ e^{-1} &\longrightarrow e \end{aligned}$$

(where notice we have reduced each term in the critical pairs to a normal form). Adding these equations introduce more critical pairs in particular:

$$\begin{aligned} x\cdot e \leftarrow \quad & x\cdot(x^{-1}\cdot(x^{-1})^{-1}) \quad && \longrightarrow (x^{-1})^{-1} \\ x\cdot(y\cdot(x\cdot y)^{-1} \leftarrow \quad & (x\cdot y)\cdot(x\cdot y)^{-1} \quad && \longrightarrow \mathsf{e} \end{aligned}$$

which causes us to add (among other things)

$$\begin{aligned} (x^{-1})^{-1} &\longrightarrow x \\ x\cdot(y\cdot(x\cdot y)^{-1} &\longrightarrow e \end{aligned}$$

This gives

$$y\cdot(x\cdot y)^{-1} \leftarrow \cdot x^{-1}\cdot(x\cdot(y\cdot(x\cdot y)^{-1}) \longrightarrow x^{-1}\cdot\mathsf{e}$$

which causes the addition of the rewrite

$$y\cdot(x\cdot y)^{-1} \longrightarrow x^{-1}$$

this rewrite gives rise to the critical pair

$$(x\cdot y)^{-1} \leftarrow y^{-1}\cdot(y\cdot(x\cdot y)^{-1}) \longrightarrow y^{-1}\cdot x^{-1}.$$

which causes the rewrite

$$(x\cdot y)^{-1} \longrightarrow y^{-1}\cdot x^{-1}$$

to be added. Finally this gives a confluent set of rewritings for groups:

$$\begin{aligned}
(x \cdot y) \cdot z &\longrightarrow x \cdot (y \cdot z) \\
\mathsf{e} \cdot x &\longrightarrow x \\
x \cdot \mathsf{e} &\longrightarrow x \\
x \cdot x^{-1} &\longrightarrow \mathsf{e} \\
x^{-1} \cdot x &\longrightarrow \mathsf{e} \\
\mathsf{e}^{-1} &\longrightarrow \mathsf{e} \\
x \cdot (x^{-1} \cdot y) &\longrightarrow y \\
x^{-1} \cdot (x \cdot y) &\longrightarrow y \\
(x^{-1})^{-1} &\longrightarrow x \\
(x \cdot y)^{-1} &\longrightarrow y^{-1} \cdot x^{-1}
\end{aligned}$$

Exercise: show that this is locally confluent and terminating.

6 Orthogonal rewriting systems

A very basic result in rewriting theory is that all left-linear orthogonal rewriting systems are confluent. A rewrite rule is **left-linear** in case the lefthand side does not contain any repeated variables (recall the problems these gave above) and contains all the variable of the righthand side. The rewriting system is **orthogonal** in case the redexes never overlap unless they are the same. An example of such a rewriting system is the system for combinatory algebra above.

Theorem 6.1 *Every left-linear orthogonal system is confluent.*

It is clear that such a system must be locally confluent, so if the system is terminating, it already implies that the system is confluent. However, proving termination can be non-trivial so that even when the system is terminating this is a useful observation. The force of this result, however, is for non-terminating systems:

Corollary 6.2 *The rewriting system on combinatory algebra is confluent.*

The remainder of this section is dedicated to proving this result in some detail. We shall use a labeling argument similar to the the one used to establish the confluence of the λ calculus.

Let $\mathcal{R} = \{r_i \longrightarrow c_i | i \in I\}$ be a left-linear orthogonal rewriting system on a set of terms $T_\Omega(X)$. Then we may augment the system with a set of labeled operations corresponding to the redexes of the rewrite rules

$$\Omega(\mathcal{R}) = \Omega \cup \{\lambda^*(x_1, ..., x_n).r_i | r_i \longrightarrow c_i \in \mathcal{R}, FV(r_i) = \{x_1, ..., x_n\}\}$$

to get a set of terms $T_{\Omega(\mathcal{R})}(X)$. In addition we may add a set of labeled rewrites corresponding to the original rewrites but starting at the labeled operations

$$\mathcal{R}^* = \mathcal{R} \cup \{(\lambda^*(x_1, ..., x_n).r_i)(x_1, ..., x_n) \longrightarrow c_i | r_i \longrightarrow c_i \in \mathcal{R}\}.$$

Thus we regard $\lambda^*(x_1,...,x_n).r_i$ as a new operation symbol which has arity n for the system which also has associated rewrite rules as above.

There is an obvious mapping $\varepsilon : T_{\Omega(*\mathcal{R})}(X) \longrightarrow T_\Omega(X)$ which simply removes the labeling. This is defined by:

$$\begin{aligned}
\varepsilon(x) &= x \qquad x \in X \\
\varepsilon(f(t_1,...,t_n)) &= f(\varphi(t_1),...,\varphi(t_n)) \\
\varepsilon((\lambda^*(x_1,...,x_n).r_i)(t_1,...,t_n)) &= r_i[\varepsilon(t_1)/x_1,...,\varepsilon(t_n)/x_n]
\end{aligned}$$

Given a term t in this labeled system we may also define a function $\varphi : T_{\Omega(\mathcal{R})}(X) \longrightarrow T_\Omega(X)$ which performs all the labeled rewrites as follows:

$$\begin{aligned}
\varphi(x) &= x \qquad x \in X \\
\varphi(f(t_1,...,t_n)) &= f(\varphi(t_1),...,\varphi(t_n)) \\
\varphi((\lambda^*(x_1,...,x_n).r_i)(t_1,...,t_n)) &= c_i[\varphi(t_1)/x_1,...,\varphi(t_n)/x_n]
\end{aligned}$$

Notice this is a "by-value" evaluation strategy.

An important observation concerning this latter function is:

Lemma 6.3 $\varphi(t[t'/x]) = \varphi(t)[\varphi(t')/x]$.

PROOF: We do a structural induction on t. There are three cases to consider:

(a) t is a variable: in this case the result is immediate.

(b) $t = f(t_1,...,t_n)$ then we have:

$$\begin{aligned}
\varphi(f(t_1,...,t_n))[\varphi(t')/x] &= f(\varphi(t_1),...,\varphi(t_n))[\varphi(t')/x] \\
&= f(\varphi(t_1)[\varphi(t')/x],...,\varphi(t_n)[\varphi(t')/x]) \\
&= f(\varphi(t_1[t'/x]),...,\varphi(t_n[t'/x])) \\
&= \varphi(f(t_1[t'/x],..,t_n[t'/x])) \\
&= \varphi(f(t_1,..,t_n)[t'/x])
\end{aligned}$$

(c) $t = (\lambda^*(x_1,...,x_n).r_1)(t_1,..,t_n)$ then we have:

$$\begin{aligned}
\varphi((\lambda^*(x_1,...,x_n).r_1)(t_1,...,t_n))[\varphi(t')/x] &= c_i[\varphi(t_1)/x_1,...,\varphi(t_n)/x_n][\varphi(t')/x] \\
&= c_i[\varphi(t_1)[\varphi(t')/x]/x_1,...,\varphi(t_n)[\varphi(t')/x]/x_n] \\
&= c_i[\varphi(t_1[t'/x])/x_1,...,\varphi(t_n[t'/x])/x_n] \\
&= c_i[\varphi(t_1[t'/x])/x_1,...,\varphi(t_n[t'/x])/x_n] \\
&= \varphi(c_i[t_1[t'/x]/x_1,...,t_n[t'/x]/x_n]) \\
&= \varphi(c_i[t_1/x_1,...,t_n/x_n][t'/x]).
\end{aligned}$$

□

This has an important consequence:

Lemma 6.4 *The following are valid rewriting diagrams:*

(i)

$$\begin{array}{ccc} r_i[t_1/x_1,...,t_n/x_n] & \xrightarrow[*]{\varphi} & \varphi(r_i[t_1/x_1,...,t_n/x_n]) \\ \Big\downarrow{\scriptstyle r_i} & & \Big\downarrow{\scriptstyle r_i} \\ c_i[t_1/x_1,...,t_n/x_n] & \xrightarrow[\varphi]{*} & \varphi(c_i[t_1/x_1,...,t_n/x_n]) \end{array}$$

(ii)

$$\begin{array}{ccc} t[r_i[t_1/x_1,...,t_n/x_n]/y] & \xrightarrow[*]{\varphi} & \varphi(t[r_i[t_1/x_1,...,t_n/x_n]/y]) \\ \Big\downarrow{\scriptstyle t[r_i/y]}* & & *\Big\downarrow{\scriptstyle \varphi(t)[r_i/y]} \\ t[c_i[t_1/x_1,...,t_n/x_n]/y] & \xrightarrow[\varphi]{*} & \varphi(t[c_i[t_1/x_1,...,t_n/x_n]/y]) \end{array}$$

(iii)

$$\begin{array}{ccc} t[(\lambda\tilde{x}.r_i)(t_1,...,t_n)/y] & & \\ \Big\downarrow{\scriptstyle t[(\lambda\tilde{x}.r_i)/y]}* & \searrow^{\varphi}_{*} & \\ t[c_i[t_1/x_1,...,t_n/x_n]/y] & \xrightarrow[\varphi]{*} & \varphi(t[(\lambda\tilde{x}.r_i)(t_1,...,t_n)/y]) \end{array}$$

PROOF:

(i) This follows as $\varphi(r_i[t_1/x_1,...,t_n/x_n]) = r_i[\varphi(t_1)/x_1,...,\varphi(t_n)/x_n]$ and $\varphi(c_i[t_1/x_1,...,t_n/x_n]) = c_i[\varphi(t_1)/x_1,...,\varphi(t_n)/x_n]$ as there is no labeled material in either r_i or c_i.

(ii) This time we make use of the above lemma as:

$$\varphi(t[r_i[t_1/x_1,...,t_n/x_n]/y]) = \varphi(t)[r_i[\varphi(t_1)/x_1,...,\varphi(t_n)/x_n]/y]$$

and

$$\varphi(t[c_i[t_1/x_1,...,t_n/x_n]/y]) = \varphi(t)[c_i[\varphi(t_1)/x_1,...,\varphi(t_n)/x_n]/y].$$

(iii) We need to show

$$\varphi(t[(\lambda^*\tilde{x}.r_i)(t_1,...,t_n)/y]) = \varphi(t[c_i(t_1,...,t_n)/y]).$$

The proof is by a structural induction on t. There are three cases:

(a) If t is a variable then either it is y and one can directly use the definition of φ or it is distinct and the result is immediate.

(b) $t = f(t'_1,...,t'_n)$ where the result holds for t'_1, ... t'_n then we have:

$$\begin{aligned} &\varphi(f(t'_1,...,t'_n)[(\lambda^*\tilde{x}.r_i)(t_1,...,t_n)/y]) \\ &\quad = \varphi(f(t'_1[(\lambda^*\tilde{x}.r_i)(t_1,...,t_n)/y],...,t'_n[(\lambda^*\tilde{x}.r_i)(t_1,...,t_n)/y])) \\ &\quad = f(\varphi(t'_1[(\lambda^*\tilde{x}.r_i)(t_1,...,t_n)/y]),...,\varphi(t_n[(\lambda^*\tilde{x}.r_i)(t_1,...,t_n)/y])) \\ &\quad = f(\varphi(t'_1)[c_i[x_1/\varphi(t_1),...,x_n/\varphi(t_n)]/y]),...,\varphi(t'_n)[c_i[\varphi(t_1)/x_1,...,\varphi(t_n))/y])) \\ &\quad = f(\varphi(t'_1)[c_i[x_1/\varphi(t_1),...,x_n/\varphi(t_n)]/y]),...,\varphi(t'_n)[c_i[\varphi(t_1)/x_1,...,\varphi(t_n))/y])) \\ &\quad = \varphi(f(t'_1,...,t'_n))[c_i[x_1/\varphi(t_1),...,x_n/\varphi(t_n)]/y] \end{aligned}$$

(c) Lastly for $t = (\lambda^* \tilde{y}.r_j)(t'_1, ..., t'_n)$ we have:

$$\begin{aligned}
&\varphi((\lambda * \tilde{y}.r_j)(t'_1, ..., t'_m)[(\lambda^* \tilde{x}.r_i)(t_1, ..., t_n)/y]) \\
&\quad = \varphi((\lambda * \tilde{y}.r_j)(t'_1[(\lambda^* \tilde{x}.r_i)(t_1, ..., t_n)/y], ..., t'_m[(\lambda^* \tilde{x}.r_i)(t_1, ..., t_n)/y])) \\
&\quad = c_j[\varphi(t'_1[(\lambda^* \tilde{x}.r_i)(t_1, ..., t_n)/y])/y_1, ..., \varphi(t'_n[(\lambda^* \tilde{x}.r_i)(t_1, ..., t_n)/y])/y_m] \\
&\quad = c_j[\varphi(t'_1)[c_i[\varphi(t_1)/x_1, ...\varphi(t_n)/x_n]/y])/y_1, ..., \varphi(t'_n)[c_i[\varphi(t_1)/x_1, ...\varphi(t_n)/x_n]/y]/y_m] \\
&\quad = \varphi(c_j[t'_1/y_1, ..., t_m/y_m])[c_i[\varphi(t_1)/x_1, ...\varphi(t_n)/x_n]/y].
\end{aligned}$$

□

Notice the above lemmas do not use the assumption that the rewrites are left linear or orthogonal.

Now suppose that we wish to determine whether a divergence of the form:

$$\begin{array}{ccc} t & \longrightarrow & t'' \\ {\scriptstyle *}\downarrow & & \\ t'' & & \end{array}$$

Then we may label the one step rewriting so that it may be viewed as $t \xrightarrow{\varphi} \varphi(t) = t''$ Note that in doing this labeling we may have to turn some of the rewritings on the multi-step leg into labeled rewritings. Here the importance of orthogonality comes into play as we must be assured that by labeling a redex we do not block a reduction but merely are forced to label it. Furthermore, introducing a label in this manner can stop a rule which is not left-linear from being applicable. So if the rules are not left-linear there is no guarantee that after labeling we can reconstruct the rewriting. Thus, at this stage we use both assumptions.

Once the labeling has been done we may argue using the induction hypothesis that when the multi-step rewriting (down) has length N or less then there is a convergence of the form:

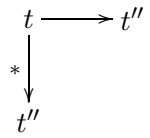
$$\begin{array}{ccc} t & \xrightarrow[*]{\varphi} & \varphi(t) \\ {\scriptstyle *}\downarrow & & \downarrow{\scriptstyle *} \\ t' & \xrightarrow[\varphi]{*} & \varphi(t') \end{array}$$

Considering an $N + 1$ step rewriting we may split it as

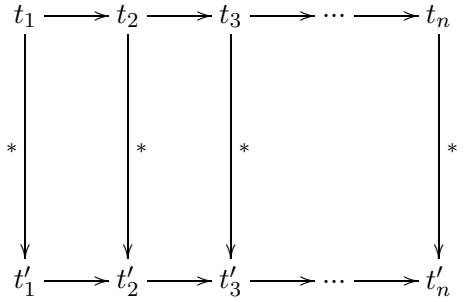
$$\begin{array}{ccc} t & \xrightarrow[*]{\varphi} & \varphi(t) \\ \downarrow & & \downarrow{\scriptstyle *} \\ t_1 & \xrightarrow[\varphi]{*} & \varphi(t_1) \\ {\scriptstyle *}\downarrow & \text{Ind.} & \downarrow{\scriptstyle *} \\ t' & \xrightarrow[\varphi]{*} & \varphi(t') \end{array}$$

where the first square comes into two flavors depending on whether the rewrite downward is a labeled or not. Both cases, however, are covered by the lemma above. Removing the labeling gives:

Lemma 6.5 (Strip lemma) *In an orthogonal left-linear system every divergence of a one-step rewriting against a multi-step rewriting has a convergence*



The main theorem now follows easily from this lemma by pasting together the strips to obtain a convergence for a divergence in which the horizontal rewrite is multi-steps as well.



7 Standard reductions

Recall that in the λ-calculus even when one has confluence that one must be careful about the order in which one reduces if one wants to be assured of finding a normal form.

We shall say that a reduction path

$$t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow .... \longrightarrow t_n$$

for an orthogonal left-linear system is **standard** if it is never the case that for $t_i \xrightarrow{r_i} t_{i+1} \xrightarrow{r_{i+1}} t_{i+2}$ the redex of r_{i+1} is above that of r_i. As the rewriting system is non-overlapping then r_i will either be parallel to r_{i+1} or completely above. Notice that if the reductions are in the wrong order in a chain that we can swap the reduction order:

$$\begin{array}{ccc} t_i = t(r_1(t_1,..,t_j(r_2(s_1,...,s_m)),...,t_n) & \xrightarrow{\quad r_2 \quad} & t(r_1(t_1,..,t_j(c_2[s_1/y_1,...,s_m/y_m]),...,t_n)) \\ \Big\downarrow{\scriptstyle r_1} & & \Big\downarrow{\scriptstyle r_1} \\ t(c_1[t_1/x_1,...,t_j(r_2(s_1,...,s_m))/x_i,...x_n/t_n]) & \xrightarrow[r_2]{*} & t(c_1[t_1/x_1,...,t_j(c_2[s_1/y_1,...,s_m/y_m])/x_i,...x_n/t_n]) \end{array}$$

Note that the left-linearity is used here and allow us to deal with an arbitrary pair of reductions. Putting the reductions in standard order can increase the number of reductions (when the variable x_i occurs more than once in c_1) but also, importantly, it can reduce the number of reductions (especially when there are infinitely many possible reductions from $t_j(c_2[s_1/y_1,...,s_m/y_m])$) as the variable x_i may not occur in c_i.

Notice that in the modified reduction chain that the maximal number of sequential reductions never increases. Two reductions are sequential (as opposed to parallel) when one occurs in a subterm of the other. This means the process of standardizing a reduction sequence will always terminate.

Proposition 7.1 *In a left-linear orthogonal rewriting system*

(i) *every reduction sequence can be transformed into a standard reduction sequence;*

(ii) *If a term has a normal form it can be obtained by a standard reduction.*

The cost of doing a standard reduction is apparently that one may have to do more reductions. However, notice that the increase in the number of reductions can be *completely* mitigated if one holds the term so that the substitutions do not replicate rewrites. This is usually done by regarding the terms as graphs so that subterms can be shared and, more importantly, the reductions on that subterm are shared (so only done once). This is essentially the idea behind graph reduction. Formally this means that these left-linear systems have an optimal rewriting strategy.

8 Combinatory algebra

As mentioned above a combinatory algebra is a model for an algebraic system with a binary operation, called application, and two constants k and s. These are subject to the following equations which we orient as rewrites:

$$\begin{aligned}(\mathsf{k} \bullet x) \bullet y &\longrightarrow x \\ ((\mathsf{s} \bullet x) \bullet y) \bullet z) &\longrightarrow ((x \bullet z) \bullet (y \bullet z)\end{aligned}$$

As we have seen above these rewrite rules form an orthogonal left-linear rewriting system. Therefore, this is a confluent system in which standard reductions are guaranteed to find normal forms. It is also a non-trivial system as $\mathsf{s}\bullet\mathsf{k} \neq \mathsf{k}$. The system, in fact, satisfies some quite remarkable properties which we now briefly explore.

An **applicative system** is a set A with a single binary operation

$$\_ \bullet \_ : A \times A \longrightarrow A; (x, y) \mapsto x \bullet y.$$

Clearly a combinatory algebra is an example of a applicative system as one can simply forget about k and s. An applicative system is said to be functionally complete if whenever there is a polynomial expression $p(x_1, ..., x_n)$ in n variable

$$p \longrightarrow x_1 | x_2 | ... | x_n | a \in A | p \bullet p$$

then there is an element $\widehat{p}$ such that for every substitution of the variables by elements of A

$$(...((\widehat{p} \bullet x_1) \bullet ...) \bullet x_n = p(x_1, ..., x_n)$$

Proposition 8.1 *Combinatory algebra is functionally complete. Furthermore, a combinatory complete applicative system is a combinatory algebra*

PROOF: The proof involves the introduction of an abstraction mechanism which is a mechanism for building constants with the desired property for functional completeness. Suppose, therefore, we have a polynomial p which involves free variables $X = x_1, ..., x_n$ and then define $\lambda^* x_i.p$ as the following polynomial in which the variable x_i has been removed:

$$\begin{array}{rcl} \lambda^* x.x & = & (\mathsf{s} \bullet \mathsf{k}) \bullet \mathsf{k} \\ \lambda^* x.z & = & \mathsf{k} \bullet z \qquad z \text{ is } \mathsf{s} \text{ or } \mathsf{k} \text{ or a variable} \\ \lambda^* x.(p_1 \bullet p_2) & = & (\mathsf{s} \bullet (\lambda^* x.p_1)) \bullet (\lambda^* x.p_2) \end{array}$$

Observe that $(\lambda * x.p) \bullet M = p[M/x]$ which can be seen by a structural induction:

$$\begin{array}{rcl} (\lambda^* x.x) \bullet M & = & ((\mathsf{s} \bullet \mathsf{k}) \bullet \mathsf{k}) \bullet M \\ & = & (\mathsf{k}) \bullet M) \bullet (\mathsf{k}) \bullet M) = M = x[M/x] \\ (\lambda^* x.z) \bullet M & = & (\mathsf{k} \bullet z) \bullet M = z = z[M/x] \\ \lambda^* x.(p_1 \bullet p_2) \bullet M & = & ((\mathsf{s} \bullet (\lambda^* x.p_1)) \bullet (\lambda^* x.p_2)) \bullet M \\ & = & ((\lambda^* x.p_1) \bullet M) \bullet ((\lambda^* x.p_2) \bullet M) \\ & = & p_1[M/x] \bullet p_2[M/x] = (p_1 \bullet p_2)[M/x] \end{array}$$

Also we observe that $(\lambda^* x.p)[M/y] = (\lambda^* x.p[M/y])$ when $x \notin FV(p)$ again a proof by strutural induction is required which we leave to the reader). This now shows that we can take

$$\widehat{p} := (\lambda^* x_1.(\lambda^* x_2....(\lambda^* x_n.p)...)$$

to obtain:

$$\begin{array}{rcl} & & ((((\lambda^* x_1.(\lambda^* x_2....(\lambda^* x_n.p)...)) \bullet M_1) \bullet M_2) \bullet ...) \bullet M_n \\ & = & (((\lambda^* x_2.(...(\lambda^* x_n.p)...)[M_1/x_1]) \bullet M_2) \bullet ...) \bullet M_n \\ & = & ((\lambda^* x_2.(...(\lambda^* x_n.p[M_1/x_1])...) \bullet M_2) \bullet ...) \bullet M_n \\ & = & p[M_1/x_1, ...M_n/x_n] \end{array}$$

For the converse it suffices to show that a combinatory algebra which is functionally complete has a k and an s. However, the equations for these combinators are functional completeness equations so this must be so! □

These observations suggest that in combinatory algebra, following the λ-calculus

$$\Omega = (\lambda^* x.x \bullet x) \bullet (\lambda^* x.x \bullet x)$$

should have a non-terminating reduction. This is easily checked to be the case. However, it should not be imagined that the two systems are equivalent: in combinatory algebra:

$$N = M \not\Rightarrow \lambda^* x.M = \lambda^* x.N$$

Thus $(\mathsf{k} \bullet x) \bullet y = x$ but

$$\begin{array}{rcl} \lambda^* x.(\mathsf{k} \bullet x) \bullet y & = & (\mathsf{s} \bullet (\lambda^* x.\mathsf{k} \bullet x)) \bullet (\lambda^* x.y) \\ & = & (\mathsf{s} \bullet ((\mathsf{s} \bullet (\lambda^* x.\mathsf{k})) \bullet (\lambda^* x.x))) \bullet (\lambda^* x.y) \\ & = & (\mathsf{s} \bullet ((\mathsf{s} \bullet (\mathsf{k} \bullet \mathsf{k})) \bullet ((\mathsf{s} \bullet \mathsf{k}) \bullet \mathsf{k}))) \bullet (\mathsf{k} \bullet y) \\ & \neq & (\mathsf{s} \bullet \mathsf{k}) \bullet \mathsf{k} = \lambda^* x.x \end{array}$$

So that combinatory logic is much weaker. Combinatory logic is important as a system as it one of the simplest systems in which all (partial) computable functions can be simulated. The encoding technique follows the techniques of the λ-calculus.

Monads for functional programming

Philip Wadler, University of Glasgow*

Department of Computing Science, University of Glasgow, G12 8QQ, Scotland
(wadler@dcs.glasgow.ac.uk)

Abstract. The use of monads to structure functional programs is described. Monads provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism. Three case studies are looked at in detail: how monads ease the modification of a simple evaluator; how monads act as the basis of a datatype of arrays subject to in-place update; and how monads can be used to build parsers.

1 Introduction

Shall I be pure or impure?

The functional programming community divides into two camps. *Pure* languages, such as Miranda[0] and Haskell, are lambda calculus pure and simple. *Impure* languages, such as Scheme and Standard ML, augment lambda calculus with a number of possible *effects*, such as assignment, exceptions, or continuations. Pure languages are easier to reason about and may benefit from lazy evaluation, while impure languages offer efficiency benefits and sometimes make possible a more compact mode of expression.

Recent advances in theoretical computing science, notably in the areas of type theory and category theory, have suggested new approaches that may integrate the benefits of the pure and impure schools. These notes describe one, the use of *monads* to integrate impure effects into pure functional languages.

The concept of a monad, which arises from category theory, has been applied by Moggi to structure the denotational semantics of programming languages [13, 14]. The same technique can be applied to structure functional programs [21, 23].

The applications of monads will be illustrated with three case studies. Section 2 introduces monads by showing how they can be used to structure a simple evaluator so that it is easy to modify. Section 3 describes the laws satisfied by

* Appears in: J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Proceedings of the Båstad Spring School, May 1995, Springer Verlag Lecture Notes in Computer Science 925. A previous version of this note appeared in: M. Broy, editor, *Program Design Calculi*, Proceedings of the Marktoberdorf Summer School, 30 July–8 August 1992. Some errata fixed August 2001; thanks to Dan Friedman for pointing these out.

[0] Miranda is a trademark of Research Software Limited.

monads. Section 4 shows how monads provide a new solution to the old problem of providing updatable state in pure functional languages. Section 5 applies monads to the problem of building recursive descent parsers; this is of interest in its own right, and because it provides a paradigm for sequencing and alternation, two of the central concepts of computing.

It is doubtful that the structuring methods presented here would have been discovered without the insight afforded by category theory. But once discovered they are easily expressed without any reference to things categorical. No knowledge of category theory is required to read these notes.

The examples will be given in Haskell, but no knowledge of that is required either. What is required is a passing familiarity with the basics of pure and impure functional programming; for general background see [3, 12]. The languages refered to are Haskell [4], Miranda [20], Standard ML [11], and Scheme [17].

2 Evaluating monads

Pure functional languages have this advantage: all flow of data is made explicit. And this disadvantage: sometimes it is painfully explicit.

A program in a pure functional language is written as a set of equations. Explicit data flow ensures that the value of an expression depends only on its free variables. Hence substitution of equals for equals is always valid, making such programs especially easy to reason about. Explicit data flow also ensures that the order of computation is irrelevant, making such programs susceptible to lazy evaluation.

It is with regard to modularity that explicit data flow becomes both a blessing and a curse. On the one hand, it is the ultimate in modularity. All data in and all data out are rendered manifest and accessible, providing a maximum of flexibility. On the other hand, it is the nadir of modularity. The essence of an algorithm can become buried under the plumbing required to carry data from its point of creation to its point of use.

Say I write an evaluator in a pure functional language.

- To add error handling to it, I need to modify each recursive call to check for and handle errors appropriately. Had I used an impure language with exceptions, no such restructuring would be needed.
- To add a count of operations performed to it, I need to modify each recursive call to pass around such counts appropriately. Had I used an impure language with a global variable that could be incremented, no such restructuring would be needed.
- To add an execution trace to it, I need to modify each recursive call to pass around such traces appropriately. Had I used an impure language that performed output as a side effect, no such restructuring would be needed.

Or I could use a *monad*.

These notes show how to use monads to structure an evaluator so that the changes mentioned above are simple to make. In each case, all that is required is to redefine the monad and to make a few local changes.

This programming style regains some of the flexibility provided by various features of impure languages. It also may apply when there is no corresponding impure feature. It does not eliminate the tension between the flexibility afforded by explicit data and the brevity afforded by implicit plumbing; but it does ameliorate it to some extent.

The technique applies not just to evaluators, but to a wide range of functional programs. For a number of years, Glasgow has been involved in constructing a compiler for the functional language Haskell. The compiler is itself written in Haskell, and uses the structuring technique described here. Though this paper describes the use of monads in a program tens of lines long, we have experience of using them in a program three orders of magnitude larger.

We begin with the basic evaluator for simple terms, then consider variations that mimic exceptions, state, and output. We analyse these for commonalities, and abstract from these the concept of a monad. We then show how each of the variations fits into the monadic framework.

2.1 Variation zero: The basic evaluator

The evaluator acts on terms, which for purposes of illustration have been taken to be excessively simple.

$$\textbf{data}\ \ \mathit{Term} = \mathit{Con}\ \mathit{Int} \mid \mathit{Div}\ \mathit{Term}\ \mathit{Term}$$

A term is either a constant $\mathit{Con}\ a$, where a is an integer, or a quotient, $\mathit{Div}\ t\ u$, where t and u are terms.

The basic evaluator is simplicity itself.

$$\begin{aligned} &\mathit{eval} && :: \mathit{Term} \to \mathit{Int} \\ &\mathit{eval}\,(\mathit{Con}\ a) && = a \\ &\mathit{eval}\,(\mathit{Div}\ t\ u) && = \mathit{eval}\ t \div \mathit{eval}\ u \end{aligned}$$

The function *eval* takes a term to an integer. If the term is a constant, the constant is returned. If the term is a quotient, its subterms are evaluated and the quotient computed. We use '$\div$' to denote integer division.

The following will provide running examples.

$$\begin{aligned} &\mathit{answer}, \mathit{error} && :: \mathit{Term} \\ &\mathit{answer} && = (\mathit{Div}\,(\mathit{Div}\,(\mathit{Con}\ \mathit{1972})\,(\mathit{Con}\ \mathit{2}))\,(\mathit{Con}\ \mathit{23})) \\ &\mathit{error} && = (\mathit{Div}\,(\mathit{Con}\ \mathit{1})\,(\mathit{Con}\ \mathit{0})) \end{aligned}$$

Computing *eval answer* yields the value of $((\mathit{1972} \div \mathit{2}) \div \mathit{23})$, which is $\mathit{42}$. The basic evaluator does not incorporate error handling, so the result of *eval error* is undefined.

2.2 Variation one: Exceptions

Say it is desired to add error checking, so that the second example above returns a sensible error message. In an impure language, this is easily achieved with the use of exceptions.

In a pure language, exception handling may be mimicked by introducing a type to represent computations that may raise an exception.

$$\begin{array}{ll} \textbf{data} & M\ a \quad\quad = \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a \\ \textbf{type} & \mathit{Exception} = \mathit{String} \end{array}$$

A value of type $M\ a$ either has the form $\mathit{Raise}\ e$, where e is an exception, or $\mathit{Return}\ a$, where a is a value of type a. By convention, a will be used both as a type variable, as in $M\ a$, and as a variable ranging over values of that type, as in $\mathit{Return}\ a$.

(A word on the difference between 'data' and 'type' declarations. A 'data' declaration introduces a new data type, in this case M, and new constructors for values of that type, in this case *Raise* and *Return*. A 'type' declaration introduces a new name for an existing type, in this case *Exception* becomes another name for *String*.)

It is straightforward, but tedious, to adapt the evaluator to this representation.

$$\begin{array}{lll} \mathit{eval} & :: & \mathit{Term} \to M\ \mathit{Int} \\ \mathit{eval}\ (\mathit{Con}\ a) & = & \mathit{Return}\ a \\ \mathit{eval}\ (\mathit{Div}\ t\ u) & = & \textbf{case}\ \ \mathit{eval}\ t\ \ \textbf{of} \\ & & \quad \mathit{Raise}\ e \to \mathit{Raise}\ e \\ & & \quad \mathit{Return}\ a \to \\ & & \quad\quad \textbf{case}\ \ \mathit{eval}\ u\ \ \textbf{of} \\ & & \quad\quad\quad \mathit{Raise}\ e \to \mathit{Raise}\ e \\ & & \quad\quad\quad \mathit{Return}\ b \to \\ & & \quad\quad\quad\quad \textbf{if}\ \ b = 0 \\ & & \quad\quad\quad\quad\quad \textbf{then}\ \ \mathit{Raise}\ \text{“}\mathit{divide\ by\ zero}\text{”} \\ & & \quad\quad\quad\quad\quad \textbf{else}\ \ \mathit{Return}\ (a \div b) \end{array}$$

At each call of the evaluator, the form of the result must be checked: if an exception was raised it is re-raised, and if a value was returned it is processed. Applying the new evaluator to *answer* yields $(\mathit{Return}\ 42)$, while applying it to *error* yields $(\mathit{Raise}\ \text{“}\mathit{divide\ by\ zero}\text{”})$.

2.3 Variation two: State

Forgetting errors for the moment, say it is desired to count the number of divisions performed during evaluation. In an impure language, this is easily achieved by the use of state. Set a given variable to zero initially, and increment it by one each time a division occurs.

In a pure language, state may be mimicked by introducing a type to represent computations that act on state.

$$\begin{array}{ll} \textbf{type} & M\ a \ = \mathit{State} \to (a, \mathit{State}) \\ \textbf{type} & \mathit{State} = \mathit{Int} \end{array}$$

Now a value of type $M\ a$ is a function that accepts the initial state, and returns the computed value paired with the final state.

Again, it is straightforward but tedious to adapt the evaluator to this representation.

$$
\begin{array}{lll}
\mathit{eval} & :: & \mathit{Term} \to M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a)\,x & = & (a, x) \\
\mathit{eval}\,(\mathit{Div}\ t\ u)\,x & = & \textbf{let}\ \ (a, y) = \mathit{eval}\ t\ x\ \ \textbf{in} \\
 & & \textbf{let}\ \ (b, z) = \mathit{eval}\ u\ y\ \ \textbf{in} \\
 & & (a \div b, z + 1)
\end{array}
$$

At each call of the evaluator, the old state must be passed in, and the new state extracted from the result and passed on appropriately. Computing *eval answer 0* yields $(42, 2)$, so with initial state *0* the answer is *42* and the final state is *2*, indicating that two divisions were performed.

2.4 Variation three: Output

Finally, say it is desired to display a trace of execution. In an impure language, this is easily done by inserting output commands at appropriate points.

In a pure language, output may be mimicked by introducing a type to represent computations that generate output.

$$
\begin{array}{lll}
\textbf{type}\ \ M\ a & = & (\mathit{Output}, a) \\
\textbf{type}\ \ \mathit{Output} & = & \mathit{String}
\end{array}
$$

Now a value of type $M\ a$ consists of the output generated paired with the value computed.

Yet again, it is straightforward but tedious to adapt the evaluator to this representation.

$$
\begin{array}{lll}
\mathit{eval} & :: & \mathit{Term} \to M\ \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a) & = & (\mathit{line}\,(\mathit{Con}\ a)\ a, a) \\
\mathit{eval}\,(\mathit{Div}\ t\ u) & = & \textbf{let}\ \ (x, a) = \mathit{eval}\ t\ \ \textbf{in} \\
 & & \textbf{let}\ \ (y, b) = \mathit{eval}\ u\ \ \textbf{in} \\
 & & (x +\!\!+ y +\!\!+ \mathit{line}\,(\mathit{Div}\ t\ u)\,(a \div b), a \div b) \\
\mathit{line} & :: & \mathit{Term} \to \mathit{Int} \to \mathit{Output} \\
\mathit{line}\ t\ a & = & \text{“}\mathit{eval}\,(\text{”} +\!\!+ \mathit{showterm}\ t +\!\!+ \text{“}) \Leftarrow \text{”} +\!\!+ \mathit{showint}\ a +\!\!+ \text{“}\hookleftarrow\text{”}
\end{array}
$$

At each call of the evaluator, the outputs must be collected and assembled to form the output of the enclosing call. The function *line* generates one line of the output. Here *showterm* and *showint* convert terms and integers to strings, $+\!\!+$ concatenates strings, and “$\hookleftarrow$” represents the string consisting of a newline.

Computing *eval answer* returns the pair $(x, 42)$, where x is the string

$$
\begin{array}{l}
\mathit{eval}\,(\mathit{Con}\ 1972) \Leftarrow 1972 \\
\mathit{eval}\,(\mathit{Con}\ 2) \Leftarrow 2 \\
\mathit{eval}\,(\mathit{Div}\,(\mathit{Con}\ 1972)\,(\mathit{Con}\ 2)) \Leftarrow 986 \\
\mathit{eval}\,(\mathit{Con}\ 23) \Leftarrow 23 \\
\mathit{eval}\,(\mathit{Div}\,(\mathit{Div}\,(\mathit{Con}\ 1972)\,(\mathit{Con}\ 2))\,(\mathit{Con}\ 23))) \Leftarrow 42
\end{array}
$$

which represents a trace of the computation.

From the discussion so far, it may appear that programs in impure languages are easier to modify than those in pure languages. But sometimes the reverse is true. Say that it was desired to modify the previous program to display the execution trace in the reverse order:

$$\begin{array}{l} \mathit{eval}\,(\mathit{Div}\,(\mathit{Div}\,(\mathit{Con}\ 1972)\,(\mathit{Con}\ 2))\,(\mathit{Con}\ 23))) \Leftarrow 42 \\ \mathit{eval}\,(\mathit{Con}\ 23) \Leftarrow 23 \\ \mathit{eval}\,(\mathit{Div}\,(\mathit{Con}\ 1972)\,(\mathit{Con}\ 2)) \Leftarrow 986 \\ \mathit{eval}\,(\mathit{Con}\ 2) \Leftarrow 2 \\ \mathit{eval}\,(\mathit{Con}\ 1972) \Leftarrow 1972 \end{array}$$

This is simplicity itself to achieve with the pure program: just replace the term

$$x +\!\!+ y +\!\!+ \mathit{line}\,(\mathit{Div}\ t\ u)\,(a \div b)$$

with the term

$$\mathit{line}\,(\mathit{Div}\ t\ u)\,(a \div b) +\!\!+ y +\!\!+ x.$$

It is not so easy to modify the impure program to achieve this effect. The problem is that output occurs as a side-effect of computation, but one now desires to display the result of computing a term *before* displaying the output generated by that computation. This can be achieved in a variety of ways, but all require substantial modification to the impure program.

2.5 A monadic evaluator

Each of the variations on the interpreter has a similar structure, which may be abstracted to yield the notion of a *monad*.

In each variation, we introduced a type of computations. Respectively, M represented computations that could raise exceptions, act on state, and generate output. By now the reader will have guessed that M stands for *monad*.

The original evaluator has the type $\mathit{Term} \to \mathit{Int}$, while in each variation its type took the form $\mathit{Term} \to M\ \mathit{Int}$. In general, a function of type $a \to b$ is replaced by a function of type $a \to M\ b$. This can be read as a function that accepts an argument of type a and returns a result of type b, with a possible additional effect captured by M. This effect may be to act on state, generate output, raise an exception, or what have you.

What sort of operations are required on the type M? Examination of the examples reveals two. First, we need a way to turn a value into the computation that returns that value and does nothing else.

$$\mathit{unit} :: a \to M\ a$$

Second, we need a way to apply a function of type $a \to M\ b$ to a computation of type $M\ a$. It is convenient to write these in an order where the argument comes before the function.

$$(\star) :: M\ a \to (a \to M\ b) \to M\ b$$

A *monad* is a triple $(M, unit, \star)$ consisting of a type constructor M and two operations of the given polymorphic types. These operations must satisfy three laws given in Section 3.

We will often write expressions in the form

$$m \star \lambda a.\, n$$

where m and n are expressions, and a is a variable. The form $\lambda a.\, n$ is a lambda expression, with the scope of the bound variable a being the expression n. The above can be read as follows: perform computation m, bind a to the resulting value, and then perform computation n. Types provide a useful guide. From the type of $(\star)$, we can see that expression m has type $M\ a$, variable a has type a, expression n has type $M\ b$, lambda expression $\lambda a.\, n$ has type $a \to M\ b$, and the whole expression has type $M\ b$.

The above is analogous to the expression

$$\textbf{let}\ \ a = m\ \ \textbf{in}\ \ n.$$

In an impure language, this has the same reading: perform computation m, bind a to the resulting value, then perform computation n and return its value. Here the types say nothing to distinguish values from computations: expression m has type a, variable a has type a, expression n has type b, and the whole expression has type b. The analogy with 'let' explains the choice of the order of arguments to $\star$. It is convenient for argument m to appear before the function $\lambda a.\, n$, since computation m is performed before computation n.

The evaluator is easily rewritten in terms of these abstractions.

$$\begin{aligned} &eval && :: Term \to M\ Int \\ &eval\,(Con\ a) && = unit\ a \\ &eval\,(Div\ t\ u) && = eval\ t \star \lambda a.\, eval\ u \star \lambda b.\, unit\,(a \div b) \end{aligned}$$

A word on precedence: lambda abstraction binds least tightly and application binds most tightly, so the last equation is equivalent to the following.

$$eval\,(Div\ t\ u) = ((eval\ t) \star (\lambda a.\, ((eval\ u) \star (\lambda b.\, (unit\,(a \div b))))))$$

The type $Term \to M\ Int$ indicates that the evaluator takes a term and performs a computation yielding an integer. To compute $(Con\ a)$, just return a. To compute $(Div\ t\ u)$, first compute t, bind a to the result, then compute u, bind b to the result, and then return $a \div b$.

The new evaluator is a little more complex than the original basic evaluator, but it is much more flexible. Each of the variations given above may be achieved by simply changing the definitions of M, $unit$, and $\star$, and by making one or two local modifications. It is no longer necessary to re-write the entire evaluator to achieve these simple changes.

2.6 Variation zero, revisited: The basic evaluator

In the simplest monad, a computation is no different from a value.

$$
\begin{array}{ll}
\textbf{type}\ \ M\ a = a & \\
\\
\mathit{unit} & :: a \to I\ a \\
\mathit{unit}\ a & = a \\
\\
(\star) & :: M\ a \to (a \to M\ b) \to M\ b \\
a \star k & = k\ a
\end{array}
$$

This is called the *identity* monad: M is the identity function on types, *unit* is the identity function, and $\star$ is just application.

Taking M, *unit*, and $\star$ as above in the monadic evaluator of Section 2.5 and simplifying yields the basic evaluator of Section 2.1

2.7 Variation one, revisited: Exceptions

In the exception monad, a computation may either raise an exception or return a value.

$$
\begin{array}{lll}
\textbf{data}\ \ M\ a & = \mathit{Raise}\ \mathit{Exception} \mid \mathit{Return}\ a & \\
\textbf{type}\ \ \mathit{Exception} & = \mathit{String} & \\
\\
\mathit{unit} & :: a \to M\ a & \\
\mathit{unit}\ a & = \mathit{Return}\ a & \\
\\
(\star) & :: M\ a \to (a \to M\ b) \to M\ b & \\
m \star k & = \textbf{case}\ \ m\ \ \textbf{of} & \\
 & \quad \mathit{Raise}\ e \to \mathit{Raise}\ e & \\
 & \quad \mathit{Return}\ a \to k\ a & \\
\\
\mathit{raise} & :: \mathit{Exception} \to M\ a & \\
\mathit{raise}\ e & = \mathit{Raise}\ e &
\end{array}
$$

The call $\mathit{unit}\ a$ simply returns the value a. The call $m \star k$ examines the result of the computation m: if it is an exception it is re-raised, otherwise the function k is applied to the value returned. Just as $\star$ in the identity monad is function application, $\star$ in the exception monad may be considered as a form of *strict* function application. Finally, the call $\mathit{raise}\ e$ raises the exception e.

To add error handling to the monadic evaluator, take the monad as above. Then just replace $\mathit{unit}\ (a \div b)$ by

$$
\begin{array}{ll}
\textbf{if}\ \ b = 0 & \\
\quad \textbf{then}\ \ \mathit{raise}\ \text{“divide by zero”} & \\
\quad \textbf{else}\ \ \mathit{unit}\ (a \div b) &
\end{array}
$$

This is commensurate with change required in an impure language.

As one might expect, this evaluator is equivalent to the evaluator with exceptions of Section 2.2. In particular, unfolding the definitions of *unit* and $\star$ in this section and simplifying yields the evaluator of that section.

2.8 Variation two, revisited: State

In the state monad, a computation accepts an initial state and returns a value paired with the final state.

$$
\begin{array}{ll}
\textbf{type}\ \ M\ a & = \mathit{State} \to (a, \mathit{State}) \\
\textbf{type}\ \ \mathit{State} & = \mathit{Int} \\
\\
\mathit{unit} & :: a \to M\ a \\
\mathit{unit}\ a & = \lambda x.\, (a, x) \\
\\
(\star) & :: M\ a \to (a \to M\ b) \to M\ b \\
m \star k & = \lambda x.\, \textbf{let}\ \ (a, y) = m\ x\ \ \textbf{in} \\
 & \qquad \textbf{let}\ \ (b, z) = k\ a\ y\ \ \textbf{in} \\
 & \qquad (b, z) \\
\\
\mathit{tick} & :: M\ () \\
\mathit{tick} & = \lambda x.\, ((), x + 1)
\end{array}
$$

The call $\mathit{unit}\ a$ returns the computation that accept initial state x and returns value a and final state x; that is, it returns a and leaves the state unchanged. The call $m \star k$ performs computation m in the initial state x, yielding value a and intermediate state y; then performs computation $k\ a$ in state y, yielding value b and final state z. The call tick increments the state, and returns the empty value $()$, whose type is also written $()$.

In an impure language, an operation like tick would be represented by a function of type $() \to ()$. The spurious argument $()$ is required to delay the effect until the function is applied, and since the output type is $()$ one may guess that the function's purpose lies in a side effect. In contrast, here tick has type $M\ ()$: no spurious argument is needed, and the appearance of M explicitly indicates what sort of effect may occur.

To add execution counts to the monadic evaluator, take the monad as above. Then just replace $\mathit{unit}\ (a \div b)$ by

$$\mathit{tick} \star \lambda().\, \mathit{unit}\ (a \div b)$$

(Here $\lambda\_.\ e$ is equivalent to $\lambda x.\ e$ where $x :: ()$ is some fresh variable that does not appear in e; it indicates that the value bound by the lambda expression must be $()$.) Again, this is commensurate with change required in an impure language. Simplifying yields the evaluator with state of Section 2.3.

2.9 Variation three, revisited: Output

In the output monad, a computation consists of the output generated paired with the value returned.

$$
\begin{array}{lll}
\textbf{type} & M\ a & = (\mathit{Output}, a) \\
\textbf{type} & \mathit{Output} & = \mathit{String} \\
\\
\mathit{unit} & & :: a \to M\ a \\
\mathit{unit}\ a & & = (\text{“”}, a) \\
\\
(\star) & & :: M\ a \to (a \to M\ b) \to M\ b \\
m \star k & & = \textbf{let}\ \ (x, a) = m\ \ \textbf{in} \\
 & & \quad \textbf{let}\ \ (y, b) = k\ a\ \ \textbf{in} \\
 & & \quad (x +\!\!+ y, b) \\
\\
\mathit{out} & & :: \mathit{Output} \to M\ () \\
\mathit{out}\ x & & = (x, ())
\end{array}
$$

The call $\mathit{unit}\ a$ returns no output paired with a. The call $m \star k$ extracts output x and value a from computation m, then extracts output y and value b from computation $k\ a$, and returns the output formed by concatenating x and y paired with the value b. The call $\mathit{out}\ x$ returns the computation with output x and empty value $()$.

To add execution traces to the monadic evaluator, take the monad as above. Then in the clause for $\mathit{Con}\ a$ replace $\mathit{unit}\ a$ by

$$\mathit{out}\ (\mathit{line}\ (\mathit{Con}\ a)\ a) \star \lambda().\ \mathit{unit}\ a$$

and in the clause for $\mathit{Div}\ t\ u$ replace $\mathit{unit}\ (a \div b)$ by

$$\mathit{out}\ (\mathit{line}\ (\mathit{Div}\ t\ u)\ (a \div b)) \star \lambda().\ \mathit{unit}\ (a \div b)$$

Yet again, this is commensurate with change required in an impure language. Simplifying yields the evaluator with output of Section 2.4.

To get the output in the reverse order, all that is required is to change the definition of $\star$, replacing $x +\!\!+ y$ by $y +\!\!+ x$. This is commensurate with the change required in the pure program, and rather simpler than the change required in an impure program.

You might think that one difference between the pure and impure versions is that the impure version displays output as it computes, while the pure version will display nothing until the entire computation completes. In fact, if the pure language is lazy then output will be displayed in an incremental fashion as the computation occurs. Furthermore, this will also happen if the order of output is reversed, which is much more difficult to arrange in an impure language. Indeed, the easiest way to arrange it may be to simulate lazy evaluation.

3 Monad laws

The operations of a monad satisfy three laws.

– *Left unit.* Compute the value a, bind b to the result, and compute n. The result is the same as n with value a substituted for variable b.

$$\mathit{unit}\ a \star \lambda b.\ n = n[a/b].$$

– *Right unit.* Compute m, bind the result to a, and return a. The result is the same as m.

$$m \star \lambda a.\ \mathit{unit}\ a = m.$$

– *Associative.* Compute m, bind the result to a, compute n, bind the result to b, compute o. The order of parentheses in such a computation is irrelevant.

$$m \star (\lambda a.\ n \star \lambda b.\ o) = (m \star \lambda a.\ n) \star \lambda b.\ o.$$

The scope of the variable a includes o on the left but excludes o on the right, so this law is valid only when a does not appear free in o.

A binary operation with left and right unit that is associative is called a *monoid*. A monad differs from a monoid in that the right operand involves a binding operation.

To demonstrate the utility of these laws, we prove that addition is associative. Consider a variant of the evaluator based on addition rather than division.

$$\begin{aligned}
&\mathbf{data}\ \ \mathit{Term} &&= \mathit{Con}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Term}\ \mathit{Term} \\
&\mathit{eval} &&:: \mathit{Term} \to M\ \mathit{Int} \\
&\mathit{eval}\ (\mathit{Con}\ a) &&= \mathit{unit}\ a \\
&\mathit{eval}\ (\mathit{Add}\ t\ u) &&= \mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{unit}\ (a \div b)
\end{aligned}$$

We show that evaluation of

$$\mathit{Add}\ t\ (\mathit{Add}\ u\ v) \quad \text{and} \quad \mathit{Add}\ (\mathit{Add}\ t\ u)\ v,$$

both compute the same result. Simplify the left term:

$$\begin{aligned}
&\mathit{eval}\ (\mathit{Add}\ t\ (\mathit{Add}\ u\ v)) \\
=\ &\{\ \text{def'n } \mathit{eval}\ \} \\
&\mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ (\mathit{Add}\ u\ v) \star \lambda x.\ \mathit{unit}\ (a + x) \\
=\ &\{\ \text{def'n } \mathit{eval}\ \} \\
&\mathit{eval}\ t \star \lambda a.\ (\mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\ (b + c)) \star \lambda x.\ \mathit{unit}\ (a + x) \\
=\ &\{\ \text{associative}\ \} \\
&\mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\ (b + c) \star \lambda x.\ \mathit{unit}\ (a + x) \\
=\ &\{\ \text{left unit}\ \} \\
&\mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\ (a + (b + c))
\end{aligned}$$

Simplify the right term similarly:

$$\begin{aligned}
&\mathit{eval}\ (\mathit{Add}\ (\mathit{Add}\ t\ u)\ v) \\
=\ &\{\ \text{as before}\ \} \\
&\mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{eval}\ v \star \lambda c.\ \mathit{unit}\ ((a + b) + c)
\end{aligned}$$

The result follows by the associativity of addition. This proof is trivial; without the monad laws, it would be impossible.

The proof works in any monad: exception, state, output. This assumes that the code is as above: if it is modified then the proof also must be modified. Section 2.3 modified the program by adding calls to *tick*. In this case, associativity still holds, as can be demonstrated using the law

$$tick \star \lambda().\, m = m \star \lambda().\, tick$$

which holds so long at *tick* is the only action on state within m. Section 2.4 modified the program by adding calls to *line*. In this case, the addition is no longer associative, in the sense that changing parentheses will change the trace, though the computations will still yield the same value.

As another example, note that for each monad we can define the following operations.

$$\begin{aligned} &map && :: (a \to b) \to (M\ a \to M\ b) \\ &map\ f\ m && = m \star \lambda a.\ unit\ (f\ a) \\ &join && :: M\ (M\ a) \to M\ a \\ &join\ z && = z \star \lambda m.\ m \end{aligned}$$

The *map* operation simply applies a function to the result yielded by a computation. To compute $map\ f\ m$, first compute m, bind a to the result, and then return $f\ a$. The *join* operation is trickier. Let z be a computation that *itself* yields a computation. To compute $join\ z$, first compute z, binds m to the result, and then behaves as computation m. Thus, *join* flattens a mind-boggling double layer of computation into a run-of-the-mill single layer of computation. As we will see in Section 5.1, lists form a monad, and for this monad *map* applies a function to each element of a list, and *join* concatenates a list of lists.

Using *id* for the identity function ($id\ x = x$), and $(\cdot)$ for function composition ($(f \cdot g)\ x = f\ (g\ x)$), one can then formulate a number of laws.

$$\begin{aligned} &map\ id && = id \\ &map\ (f \cdot g) && = map\ f \cdot map\ g \\ &map\ f \cdot unit && = unit \cdot f \\ &map\ f \cdot join && = join \cdot map\ (map\ f) \\ &join \cdot unit && = id \\ &join \cdot map\ unit && = id \\ &join \cdot map\ join && = join \cdot join \\ &m \star k && = join\ (map\ k\ m) \end{aligned}$$

The proof of each is a simple consequence of the definitions of *map* and *join* and the three monad laws.

Often, monads are defined not in terms of *unit* and $\star$, but rather in terms of *unit*, *join*, and *map* [10, 13]. The three monad laws are replaced by the first seven of the eight laws above. If one defines $\star$ by the eighth law, then the three monad laws follow. Hence the two definitions are equivalent.

4 State

Arrays play a central role in computing because they closely match current architectures. Programs are littered with array lookups such as $x[i]$ and array updates such as $x[i] := v$. These operations are popular because array lookup is implemented by a single indexed fetch instruction, and array update by a single indexed store.

It is easy to add arrays to a functional language, and easy to provide efficient array lookup. How to provide efficient array update, on the other hand, is a question with a long history. Monads provide a new answer to this old question.

Another question with a long history is whether it is *desirable* to base programs on array update. Since so much effort has gone into developing algorithms and architectures based on arrays, we will sidestep this debate and simply assume the answer is yes.

There is an important difference between the way monads are used in the previous section and the way monads are used here. The previous section showed monads help to use existing language features more effectively; this section shows how monads can help define new language features. No change to the programming language is required, but the implementation must provide a new abstract data type, perhaps as part of the standard prelude.

Here monads are used to manipulate state internal to the program, but the same techniques can be use to manipulate extenal state: to perform input/output, or to communicate with other programming languages. The Glasgow implementation of Haskell uses a design based on monads to provide input/output and interlanguage working with the imperative programming language C [15]. This design has been adopted for version 1.3 of the Haskell standard.

4.1 Arrays

Let Arr be the type of arrays taking indexes of type Ix and yielding values of type Val. The key operations on this type are

$$\begin{aligned} &\mathit{newarray} :: \mathit{Val} \to \mathit{Arr}, \\ &\mathit{index} \quad\;\; :: \mathit{Ix} \to \mathit{Arr} \to \mathit{Val}, \\ &\mathit{update} \quad :: \mathit{Ix} \to \mathit{Val} \to \mathit{Arr} \to \mathit{Arr}. \end{aligned}$$

The call $\mathit{newarray}\ v$ returns an array with all entries set to v; the call $\mathit{index}\ i\ x$ returns the value at index i in array x; and the call $\mathit{update}\ i\ v\ x$ returns an array where index i has value v and the remainder is identical to x. The behaviour of these operations is specified by the laws

$$\begin{aligned} &\mathit{index}\ i\ (\mathit{newarray}\ v) = v, \\ &\mathit{index}\ i\ (\mathit{update}\ i\ v\ x) = v, \\ &\mathit{index}\ i\ (\mathit{update}\ j\ v\ x) = \mathit{index}\ i\ x, \text{ if } i \neq j. \end{aligned}$$

In practice, these operations would be more complex; one needs a way to specify the index bounds, for instance. But the above suffices to explicate the main points.

The efficient way to implement the update operation is to overwrite the specified entry of the array, but in a pure functional language this is only safe if there are no other pointers to the array extant when the update operation is performed. An array satisfying this property is called *single threaded*, following Schmidt [18].

Consider building an interpreter for a simple imperative language. The abstract syntax for this language is represented by the following data types.

$$\begin{array}{l}\textbf{data}\ \ \mathit{Term} \ \ = \mathit{Var\ Id} \mid \mathit{Con\ Int} \mid \mathit{Add\ Term\ Term} \\ \textbf{data}\ \ \mathit{Comm} = \mathit{Asgn\ Id\ Term} \mid \mathit{Seq\ Comm\ Comm} \mid \mathit{If\ Term\ Comm\ Comm} \\ \textbf{data}\ \ \mathit{Prog} \ \ = \mathit{Prog\ Comm\ Term}\end{array}$$

Here $Id : baastad.tex, v1.1.1.11996/02/2915 : 17 : 01wadlerExp$ is an unspecified type of identifiers. A term is a variable, a constant, or the sum of two terms; a command is an assignment, a sequence of two commands, or a conditional; and a program consists of a command followed by a term.

The current state of execution will be modelled by an array where the indexes are identifiers and the corresponding values are integers.

$$\begin{array}{l}\textbf{type}\ \ \mathit{State} = \mathit{Arr} \\ \textbf{type}\ \ \mathit{Ix} \ \ \ \ = \mathit{Id} \\ \textbf{type}\ \ \mathit{Val} \ \ \ = \mathit{Int}\end{array}$$

Here is the interpreter.

$$\begin{array}{lll}\mathit{eval} & :: & \mathit{Term} \to \mathit{State} \to \mathit{Int} \\ \mathit{eval}\,(\mathit{Var}\ i)\,x & = & \mathit{index}\ i\ x \\ \mathit{eval}\,(\mathit{Con}\ a)\,x & = & a \\ \mathit{eval}\,(\mathit{Add}\ t\ u)\,x & = & \mathit{eval}\ t\ x + \mathit{eval}\ u\ x \\ \\ \mathit{exec} & :: & \mathit{Comm} \to \mathit{State} \to \mathit{State} \\ \mathit{exec}\,(\mathit{Asgn}\ i\ t)\,x & = & \mathit{update}\ i\,(\mathit{eval}\ t\ x)\,x \\ \mathit{exec}\,(\mathit{Seq}\ c\ d)\,x & = & \mathit{exec}\ d\,(\mathit{exec}\ c\ x) \\ \mathit{exec}\,(\mathit{If}\ t\ c\ d)\,x & = & \textbf{if}\ \ \mathit{eval}\ t\ x = 0\ \ \textbf{then}\ \ \mathit{exec}\ c\ x\ \ \textbf{else}\ \ \mathit{exec}\ d\ x \\ \\ \mathit{elab} & :: & \mathit{Prog} \to \mathit{Int} \\ \mathit{elab}\,(\mathit{Prog}\ c\ t) & = & \mathit{eval}\ t\,(\mathit{exec}\ c\,(\mathit{newarray}\ 0))\end{array}$$

This closely resembles a denotational semantics. The evaluator for terms takes a term and a state and returns an integer. Evaluation of a variable is implemented by indexing the state. The executor for commands takes a command and the initial state and returns the final state. Assignment is implemented by updating the state. The elaborator for programs takes a program and returns an integer. It executes the command in an initial state where all identifiers map to 0, then evaluates the given expression in the resulting state and returns its value.

The state in this interpreter is single threaded: at any moment of execution there is only one pointer to the state, so it is safe to update the state in place. In order for this to work, the update operation must evaluate the new value before placing it in the array. Otherwise, the array may contain a closure that itself contains a pointer to the array, violating the single threading property. In semantic terms, one says that *update* is strict in all three of its arguments.

A number of researchers have proposed analyses that determine whether a given functional program uses an array in a single threaded manner, with the intent of incorporating such an analysis into an optimising compiler. Most of these analyses are based on abstract interpretation [1]. Although there has been some success in this area, the analyses tend to be so expensive as to be intractable [2, 7].

Even if such analyses were practicable, their use may be unwise. Optimising update can affect a program's time and space usage by an order of magnitude or more. The programmer must be assured that such an optimisation will occur in order to know that the program will run adequately fast and within the available space. It may be better for the programmer to indicate explicitly that an array should be single threaded, rather than leave it to the vagaries of an optimising compiler.

Again, a number of researchers have proposed techniques for indicating that an array is single threaded. Most of these techniques are based on type systems [6, 19, 22]. This area seems promising, although the complexities of these type systems remain formidable.

The following section presents another way of indicating explicitly the intention that an array be single threaded. Naturally, it is based on monads. The advantage of this method is that it works with existing type systems, using only the idea of an abstract data type.

4.2 Array transformers

The monad of array transformers is simply the monad of state transformers, with the state taken to be an array. The definitions of M, *unit*, $\star$ are as before.

$$
\begin{array}{ll}
\textbf{type}\ \ M\ a & = \mathit{State} \to (a, \mathit{State}) \\
\textbf{type}\ \ \mathit{State} & = \mathit{Arr} \\
\\
\mathit{unit} & :: a \to M\ a \\
\mathit{unit}\ a & = \lambda x.\, (a, x) \\
\\
(\star) & :: M\ a \to (a \to M\ b) \to M\ b \\
m \star k & = \lambda x.\, \textbf{let}\ \ (a, y) = m\ x\ \ \textbf{in} \\
 & \qquad\quad \textbf{let}\ \ (b, z) = k\ a\ y\ \ \textbf{in} \\
 & \qquad\quad (b, z)
\end{array}
$$

Previously, our state was an integer and we had an additional operation *tick* acting upon the state. Now our state is an array, and we have additional operations

corresponding to array creation, indexing, and update.

$$
\begin{array}{ll}
\mathit{block} & :: \mathit{Val} \to M\ a \to a \\
\mathit{block}\ v\ m & = \mathbf{let}\ \ (a, x) = m\,(\mathit{newarray}\ v)\ \ \mathbf{in}\ \ a \\
\mathit{fetch} & :: \mathit{Ix} \to M\ \mathit{Val} \\
\mathit{fetch}\ i & = \lambda x.\,(\mathit{index}\ i\ x, x) \\
\mathit{assign} & :: \mathit{Ix} \to \mathit{Val} \to M\ () \\
\mathit{assign}\ i\ v & = \lambda x.\,((), \mathit{update}\ i\ v\ x)
\end{array}
$$

The call $\mathit{block}\ v\ m$ creates a new array with all locations initialised to v, applies monad m to this initial state to yield value a and final state x, deallocates the array, and returns a. The call $\mathit{fetch}\ i$ returns the value at index i in the current state, and leaves the state unchanged. The call $\mathit{assign}\ i\ v$ returns the empty value $()$, and updates the state so that index i contains value v.

A little thought shows that these operations are indeed single threaded. The only operation that could duplicate the array is *fetch*, but this may be implemented as follows: first fetch the entry at the given index in the array, and then return the pair consisting of this value and the pointer to the array. In semantic terms, *fetch* is strict in the array and the index, but not in the value located at the index, and *assign* is strict in the array and the index, but not the value assigned.

(This differs from the previous section, where in order for the interpreter to be single threaded it was necessary for *update* to be strict in the given value. In this section, as we shall see, this strictness is removed but a spurious sequencing is introduced for evaluation of terms. In the following section, the spurious sequencing is removed, but the strictness will be reintroduced.)

We may now make M into an *abstract data type* supporting the five operations *unit*, $\star$, *block*, *fetch*, and *assign*. The operation *block* plays a central role, as it is the only one that does not have M in its result type. Without *block* there would be no way to write a program using M that did not have M in its output type.

Making M into an abstract data type guarantees that single threading is preserved, and hence it is safe to implement assignment with an in-place update. The use of data abstraction is essential for this purpose. Otherwise, one could write programs such as

$$\lambda x.\,(\mathit{assign}\ i\ v\ x, \mathit{assign}\ i\ w\ x)$$

that violate the single threading property.

The interpreter may now be rewritten as follows.

$$
\begin{array}{ll}
\mathit{eval} & :: \mathit{Term} \to M\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Var}\ i) & = \mathit{fetch}\ i \\
\mathit{eval}\ (\mathit{Con}\ a) & = \mathit{unit}\ a \\
\mathit{eval}\ (\mathit{Add}\ t\ u) & = \mathit{eval}\ t \star \lambda a.\ \mathit{eval}\ u \star \lambda b.\ \mathit{unit}\ (a + b) \\
\\
\mathit{exec} & :: \mathit{Comm} \to M\ () \\
\mathit{exec}\ (\mathit{Asgn}\ i\ t) & = \mathit{eval}\ t \star \lambda a.\ \mathit{assign}\ i\ a \\
\mathit{exec}\ (\mathit{Seq}\ c\ d) & = \mathit{exec}\ c \star \lambda().\ \mathit{exec}\ d \star \lambda().\ \mathit{unit}\ () \\
\mathit{exec}\ (\mathit{If}\ t\ c\ d) & = \mathit{eval}\ t \star \lambda a. \\
 & \quad \textbf{if}\ \ a == 0\ \ \textbf{then}\ \ \mathit{exec}\ c\ \ \textbf{else}\ \ \mathit{exec}\ d \\
\\
\mathit{elab} & :: \mathit{Prog} \to \mathit{Int} \\
\mathit{elab}\ (\mathit{Prog}\ c\ t) & = \mathit{block}\ 0\ (\mathit{exec}\ c \star \lambda().\ \mathit{eval}\ t \star \lambda a.\ \mathit{unit}\ a)
\end{array}
$$

The types show that evaluation of a term returns an integer and may access or modify the state, and that execution of a term returns nothing and may access or modify the state. In fact, evaluation only accesses the state and never alters it — we will consider shortly a more refined system that allows us to indicate this.

The abstract data type for M guarantees that it is safe to perform updates in place – no special analysis technique is required. It is easy to see how the monad interpreter can be derived from the original, and (using the definitions given earlier) the proof of their equivalence is straightforward.

The rewritten interpreter is slightly longer than the previous version, but perhaps slightly easier to read. For instance, execution of $(\mathit{Seq}\ c\ d)$ can be read: compute the execution of c, then compute the execution of d, then return nothing. Compare this with the previous version, which has the unnerving property that *exec d* appears to the left of *exec c*.

One drawback of this program is that it introduces too much sequencing. Evaluation of $(\mathit{Add}\ t\ u)$ can be read: compute the evaluation of t, bind a to the result, then compute the evaluation of u, bind b to the result, then return $a + b$. This is unfortunate, in that it imposes a spurious ordering on the evaluation of t and u that was not present in the original program. The order does not matter because although *eval* depends on the state, it does not change it. To remedy this we will augment the monad of state transformers M with a second monad M' of state readers.

4.3 Array readers

Recall that the monad of array transformers takes an initial array and returns a value and a final array.

$$
\begin{array}{ll}
\textbf{type} & M\ a\ = \mathit{State} \to (a, \mathit{State}) \\
\textbf{type} & \mathit{State} = \mathit{Arr}
\end{array}
$$

The corresponding monad of array readers takes an array and returns a value. No array is returned because it is assumed identical to the original array.

$$\begin{array}{ll}
\textbf{type} & M'\ a = \mathit{State} \to a \\
\mathit{unit}' & :: a \to M'\ a \\
\mathit{unit}'\ a & = \lambda x.\ a \\
(\star') & :: M'\ a \to (a \to M'\ b) \to M'\ b \\
m \star' k & = \lambda x.\ \textbf{let}\ \ a = m\ x\ \ \textbf{in}\ \ k\ a\ x \\
\mathit{fetch}' & :: \mathit{Ix} \to M'\ \mathit{Val} \\
\mathit{fetch}'\ i & = \lambda x.\ \mathit{index}\ i\ x
\end{array}$$

The call $\mathit{unit}'\ a$ ignores the given state x and returns a. The call $m \star' k$ performs computation m in the given state x, yielding value a, then performs computation $k\ a$ in the same state x. Thus, unit' discards the state and $\star'$ duplicates it. The call $\mathit{fetch}'\ i$ returns the value in the given state x at index i.

Clearly, computations that only read the state are a subset of the computations that may read and write the state. Hence there should be a way to coerce a computation in monad M' into one in monad M.

$$\begin{array}{ll}
\mathit{coerce} & :: M'\ a \to M\ a \\
\mathit{coerce}\ m & = \lambda x.\ \textbf{let}\ \ a = m\ x\ \ \textbf{in}\ \ (a, x)
\end{array}$$

The call $\mathit{coerce}\ m$ performs computation m in the initial state x, yielding a, and returns a paired with state x. The function *coerce* enjoys a number of mathematical properties to be discussed shortly.

Again, these operations maintain single threading if suitably implemented. The definitions of $\star'$ and *coerce* must both be strict in the intermediate value a. This guarantees that when $\mathit{coerce}\ m$ is performed in state x, the computation of $m\ x$ will reduce to a form a that contains no extant pointers to the state x before the pair (a, x) is returned. Hence there will be only one pointer extant to the state whenever it is updated.

A monad is *commutative* if it satisfies the law

$$m \star \lambda a.\ n \star \lambda b.\ o = n \star \lambda b.\ m \star \lambda a.\ o.$$

The scope of a includes n on the right and not on the left, so this law is valid only when a does not appear free in n. Similarly, b must not appear free in m. In a commutative monad the order of computation does not matter.

The state reader monad is commutative, while the state transfomer monad is not. So no spurious order is imposed on computations in the state reader monad. In particular, the call $m \star' k$ may safely be implemented so that m and $k\ a$ are computed in parallel. However, the final result must still be strict in a. For instance, with the annotations used in the GRIP processor, $\star'$ could be defined as follows.

$$\begin{array}{rl}
m \star' k = \lambda x. & \textbf{let}\ \ a = m\ x\ \ \textbf{in} \\
& \textbf{let}\ \ b = k\ a\ x\ \ \textbf{in} \\
& \mathit{par}\ a\ (\mathit{par}\ b\ (\mathit{seq}\ a\ b))
\end{array}$$

The two calls to *par* spark parallel computations of a and b, and the call to *seq* waits for a to reduce to a non-bottom value before returning b.

These operations may be packaged into two abstract data types, M and M', supporting the eight operations $unit$, $\star$, $unit'$, $\star'$, $block$, $assign$, $fetch'$, and $coerce$. The abstraction guarantees single threading, so $assign$ may be implemented by an in-place update.

The interpreter may be rewritten again.

$$
\begin{array}{lll}
eval & :: & Term \to M'\ Int \\
eval\ (Var\ i) & = & fetch'\ i \\
eval\ (Con\ a) & = & unit'\ a \\
eval\ (Add\ t\ u) & = & eval\ t \star' \lambda a.\ eval\ u \star' \lambda b.\ unit'\ (a + b) \\
 & & \\
exec & :: & Comm \to M\ () \\
exec\ (Asgn\ i\ t) & = & coerce\ (eval\ t) \star \lambda a.\ assign\ i\ a \\
exec\ (Seq\ c\ d) & = & exec\ c \star \lambda().\ exec\ d \star \lambda().\ unit\ () \\
exec\ (If\ t\ c\ d) & = & coerce\ (eval\ t) \star \lambda a. \\
 & & \quad \textbf{if}\ \ a = 0\ \ \textbf{then}\ \ exec\ c\ \ \textbf{else}\ \ exec\ d \\
 & & \\
elab & :: & Prog \to Int \\
elab\ (Prog\ c\ t) & = & block\ 0\ (exec\ c \star \lambda().\ coerce\ (eval\ t) \star \lambda a.\ unit\ a)
\end{array}
$$

This differs from the previous version in that *eval* is written in terms of M' rather than M, and calls to *coerce* surround the calls of *eval* in the other two functions. The new types make it clear that *eval* depends upon the state but does not alter it, while *exec* may both depend upon and alter the state.

The excessive sequencing of the previous version has been eliminated. In the evaluation of $(Add\ t\ u)$ the two subexpressions may be evaluated in either order or concurrently.

A *monad morphism* from a monad M' to a monad M is a function $h :: M'\ a \to M\ a$ that preserves the monad structure:

$$
\begin{array}{lll}
h\ (unit'\ a) & = & unit\ a, \\
h\ (m \star' \lambda a.\ n) & = & (h\ m) \star \lambda a.\ (h\ n).
\end{array}
$$

It often happens that one wishes to use a combination of monads to achieve a purpose, and monad morphisms play the key role of converting from one monad to another [9].

In particular, *coerce* is a monad morphism, and it follows immediately from this that the two versions of the interpreter are equivalent.

4.4 Conclusion

How a functional language may provide in-place array update is an old problem. This section has presented a new solution, consisting of two abstract data types with eight operations between them. No change to the programming language is required, other than to provide an implementation of these types, perhaps as part of the standard prelude. The discovery of such a simple solution comes as

a surpise, considering the plethora of more elaborate solutions that have been proposed.

A different way of expressing the same solution, based on continuation passing style, has subsequently been proposed by Hudak [8]. But Hudak's solution was inspired by the monad solution, and the monad solution still appears to have some small advantages [15].

Why was this solution not discovered twenty years ago? One possible reason is that the data types involve higher-order functions in an essential way. The usual axiomatisation of arrays involves only first-order functions, and so perhaps it did not occur to anyone to search for an abstract data type based on higher-order functions. That monads led to the discovery of the solution must count as a point in their favour.

5 Parsers

Parsers are the great success story of theoretical computing. The BNF formalism provides a concise and precise way to describe the syntax of a programming language. Mathematical tests can determine if a BNF grammar is ambiguous or vacuous. Transformations can produce an equivalent grammar that is easier to parse. Compiler-compilers can turn a high-level specification of a grammar into an efficient program.

This section shows how monads provide a simple framework for constructing recursive descent parsers. This is of interest in its own right, and also because the basic structures of parsing – sequencing and alternation – are fundamental to all of computing. It also provides a demonstration of how monads can model backtracking (or angelic non-determinism).

5.1 Lists

Our representation of parsers depends upon lists. Lists are ubiquitous in functional programming, and it is surprising that we have managed to get by so far while barely mentioning them. Actually, they have appeared in disguise, as strings are simply lists of characters.

We review some notation. We write $[a]$ for the type of a list with elements all of type a, and : for 'cons'. Thus $[1, 2, 3] = 1 : 2 : 3 : [\,]$, and both have type $[Int]$. Strings are lists of characters, so $String$ and $[Char]$ are equivalent, and $\text{"}monad\text{"}$ is just an abbreviation for $[\text{'}m\text{'}, \text{'}o\text{'}, \text{'}n\text{'}, \text{'}a\text{'}, \text{'}d\text{'}]$.

It is perhaps not suprising that lists form a monad.

$$
\begin{array}{ll}
unit & :: a \to [a] \\
unit\ a & = [a] \\
(\star) & :: [a] \to (a \to [b]) \to [b] \\
[\,] \star k & = [\,] \\
(a : x) \star k & = k\ a +\!\!+ (x \star k)
\end{array}
$$

The call *unit a* simply forms the unit list containing a. The call $m \star k$ applies k to each element of the list m, and appends together the resulting lists.

If monads encapsulate effects and lists form a monad, do lists correspond to some effect? Indeed they do, and the effect they correspond to is choice. One can think of a computation of type $[a]$ as offering a choice of values, one for each element of the list. The monadic equivalent of a function of type $a \to b$ is a function of type $a \to [b]$. This offers a choice of results for each argument, and hence corresponds to a relation. The operation *unit* corresponds to the identity relation, which associates each argument only with itself. If $k :: a \to [b]$ and $h :: b \to [c]$, then

$$\lambda a.\, k\ a \star \lambda b.\, h\ b :: a \to [c]$$

corresponds to the relational composition of k and h.

The *list comprehension* notation provides a convenient way of manipulating lists. The behaviour is analogous to set comprehensions, except the order is significant. For example,

$$\begin{array}{ll} [\, sqr\ a \mid a \leftarrow [1,2,3]\,] & = [1,4,9] \\ [\, (a,b) \mid a \leftarrow [1,2], b \leftarrow \text{“list”}\,] & = [(1,\text{‘l’}),(1,\text{‘i’}),(1,\text{‘s’}),(1,\text{‘t’}), \\ & \quad (2,\text{‘l’}),(2,\text{‘i’}),(2,\text{‘s’}),(2,\text{‘t’})] \end{array}$$

The list comprehension notation translates neatly into monad operations.

$$\begin{array}{ll} [\, t \mid x \leftarrow u\,] & = u \star \lambda x.\, unit\ t \\ [\, t \mid x \leftarrow u,\ y \leftarrow v\,] & = u \star \lambda x.\, v \star \lambda y.\, unit\ t \end{array}$$

Here t is an expression, x and y are variables (or more generally patterns), and u and v are expressions that evaluate to lists. Connections between comprehensions and monads have been described at length elsewhere [21].

5.2 Representing parsers

Parsers are represented in a way similar to state transformers.

$$\begin{array}{ll} \textbf{type} & M\ a\ = State \to [(a, State)] \\ \textbf{type} & State = String \end{array}$$

That is, the parser for type a takes a state representing a string to be parsed, and returns a *list* of containing the value of type a parsed from the string, and a state representing the remaining string yet to be parsed. The list represents all the alternative possible parses of the input state: it will be empty if there is no parse, have one element if there is one parse, have two elements if there are two different possible parses, and so on.

Consider a simple parser for arithmetic expressions, which returns a tree of the type considered previously.

$$\textbf{data}\ \ Term = Con\ Int \mid Div\ Term\ Term$$

Say we have a parser for such terms.

$$term :: M\ Term$$

Here are some examples of its use.

$$\begin{array}{ll} term\ \text{“23”} & = [(Con\ 23, \text{“”})] \\ term\ \text{“23 and more”} & = [(Con\ 23, \text{“ and more”})] \\ term\ \text{“not a term”} & = [\,] \\ term\ \text{“((1972 ÷ 2) ÷ 23)”} & = [(Div\ (Div\ (Con\ 1972)\ (Con\ 2))\ (Con\ 23)), \text{“”})] \end{array}$$

A parser m is *unambiguous* if for every input x the list of possible parses $m\ x$ is either empty or has exactly one item. For instance, *term* is unambiguous. An ambiguous parser may return a list with two or more alternative parsings.

5.3 Parsing an item

The basic parser returns the first item of the input, and fails if the input is exhausted.

$$\begin{array}{ll} item & :: M\ Char \\ item\ [\,] & = [\,] \\ item\ (a : x) & = [(a, x)] \end{array}$$

Here are two examples.

$$\begin{array}{ll} item\ \text{“”} & = [\,] \\ item\ \text{“monad”} & = [(\text{‘m’}, \text{“onad”})] \end{array}$$

Clearly, *item* is unambiguous.

5.4 Sequencing

To form parsers into a monad, we require operations *unit* and $\star$.

$$\begin{array}{ll} unit & :: a \to M\ a \\ unit\ a\ x & = [(a, x)] \\ (\star) & :: M\ a \to (a \to M\ b) \to M\ b \\ (m \star k)\ x & = [(b, z) \mid (a, y) \leftarrow m\ x,\ (b, z) \leftarrow k\ a\ y] \end{array}$$

The parser *unit a* accepts input x and yields one parse with value a paired with remaining input x. The parser $m \star k$ takes input x; parser m is applied to input x yielding for each parse a value a paired with remaining input y; then parser $k\ a$ is applied to input y, yielding for each parse a value b paired with final remaining output z.

Thus, *unit* corresponds to the empty parser, which consumes no input, and $\star$ corresponds to sequencing of parsers.

Two items may be parsed as follows.

$$\begin{array}{l} twoItems :: M\ (Char, Char) \\ twoItems = item \star \lambda a.\ item \star \lambda b.\ unit\ (a, b) \end{array}$$

Here are two examples.

$$\begin{aligned} &\mathit{twoItems}\ \text{“}m\text{”} &&= [\,] \\ &\mathit{twoItems}\ \text{“}\mathit{monad}\text{”} &&= [((\text{‘}m\text{’}, \text{‘}o\text{’}), \text{“}\mathit{nad}\text{”})] \end{aligned}$$

The parse succeeds only if there are at least two items in the list.

The three monad laws express that the empty parser is an identity for sequencing, and that sequencing is associative.

$$\begin{aligned} \mathit{unit}\ a \star \lambda b.\ n &= n[a/b] \\ m \star \lambda a.\ \mathit{unit}\ a &= m \\ m \star (\lambda a.\ n \star \lambda b.\ o) &= (m \star \lambda a.\ n) \star \lambda b.\ o \end{aligned}$$

If m is unambiguous and $k\ a$ is unambiguous for every a, then $m \star k$ is also unambiguous.

5.5 Alternation

Parsers may also be combined by alternation.

$$\begin{aligned} &\mathit{zero} &&:: M\ a \\ &\mathit{zero}\ x &&= [\,] \\ &(\oplus) &&:: M\ a \to M\ a \to M\ a \\ &(m \oplus n)\ x &&= m\ x +\!\!+ n\ x \end{aligned}$$

The parser zero takes input x and always fails. The parser $m \oplus n$ takes input x and yields all parses of m applied to input x and all parses of n applied to the same input x.

Here is a parser that parses one or two items from the input.

$$\begin{aligned} &\mathit{oneOrTwoItems} :: M\ \mathit{String} \\ &\mathit{oneOrTwoItems} = \quad (\mathit{item} \star \lambda a.\ \mathit{unit}\,[a]) \\ &\qquad\qquad\qquad\quad \oplus\, (\mathit{item} \star \lambda a.\ \mathit{item} \star \lambda b.\ \mathit{unit}\,[a, b]) \end{aligned}$$

Here are three examples.

$$\begin{aligned} &\mathit{oneOrTwoItems}\ \text{“”} &&= [\,] \\ &\mathit{oneOrTwoItems}\ \text{“}m\text{”} &&= [(\text{“}m\text{”}, \text{“”})] \\ &\mathit{oneOrTwoItems}\ \text{“}\mathit{monad}\text{”} &&= [(\text{“}m\text{”}, \text{“}\mathit{onad}\text{”}), (\text{“}\mathit{mo}\text{”}, \text{“}\mathit{nad}\text{”})] \end{aligned}$$

The last yields two alternative parses, showing that alternation can yield ambiguous parsers.

The parser that always fails is the identity for alternation, and alternation is associative.

$$\begin{aligned} \mathit{zero} \oplus n &= n \\ m \oplus \mathit{zero} &= m \\ m \oplus (n \oplus o) &= (m \oplus n) \oplus o \end{aligned}$$

Furthermore, *zero* is indeed a zero of $\star$, and $\star$ distributes through $\oplus$.

$$\begin{aligned} \mathit{zero} \star k &= \mathit{zero} \\ m \star \lambda a.\, \mathit{zero} &= \mathit{zero} \\ (m \oplus n) \star k &= (m \star k) \oplus (n \star k) \end{aligned}$$

It is *not* the case that $\star$ distributes rightward through $\oplus$ only because we are representing alternative parses by an ordered list; if we used an unordered bag, then $m \star \lambda a.\, (k\ a \oplus h\ a) = (m \star k) \oplus (m \star h)$ would also hold. An unambiguous parser yields a list of length at most one, so the order is irrelevant, and hence this law also holds whenever either side is unambiguous (which implies that both sides are).

5.6 Filtering

A parser may be filtered by combining it with a predicate.

$$\begin{aligned} (\rhd) &:: M\ a \to (a \to \mathit{Bool}) \to M\ a \\ m \rhd p &= m \star \lambda a.\, \mathbf{if}\ \ p\ a\ \ \mathbf{then}\ \ \mathit{unit}\ a\ \ \mathbf{else}\ \ \mathit{zero} \end{aligned}$$

Given a parser m and a predicate on values p, the parser $m \rhd p$ applies parser m to yield a value a; if $p\ a$ holds it succeeds with value a, otherwise it fails. Note that filtering is written in terms of previously defined operators, and need not refer directly to the state.

Let *isLetter* and *isDigit* be the obvious predicates. Here are two parsers.

$$\begin{aligned} &\mathit{letter} :: M\ \mathit{Char} \\ &\mathit{letter} = \mathit{item} \rhd \mathit{isLetter} \end{aligned}$$

$$\begin{aligned} &\mathit{digit} :: M\ \mathit{Int} \\ &\mathit{digit} = (\mathit{item} \rhd \mathit{isDigit}) \star \lambda a.\, \mathit{unit}\,(\mathit{ord}\ a - \mathit{ord}\ \text{`0'}) \end{aligned}$$

The first succeeds only if the next input item is a letter, and the second succeeds only if it is a digit. The second also converts the digit to its corresponding value, using $\mathit{ord} :: \mathit{Char} \to \mathit{Int}$ to convert a character to its ASCII code. Assuming that $\rhd$ has higher precedence than $\star$ would allow some parentheses to be dropped from the second definition.

A parser for a literal recognises a single specified character.

$$\begin{aligned} \mathit{lit} &:: \mathit{Char} \to M\ \mathit{Char} \\ \mathit{lit}\ c &= \mathit{item} \rhd (\lambda a.\, a = c) \end{aligned}$$

The parser *lit c* succeeds if the input begins with character c, and fails otherwise.

$$\begin{aligned} \mathit{lit}\ \text{`m'}\ \text{“monad”} &= [(\text{`m'}, \text{“onad”})] \\ \mathit{lit}\ \text{`m'}\ \text{“parse”} &= [\,] \end{aligned}$$

From the previous laws, it follows that filtering preserves zero and distributes through alternation.

$$\begin{aligned} \mathit{zero} \rhd p &= \mathit{zero} \\ (m \oplus n) \rhd p &= (m \rhd p) \oplus (n \rhd p) \end{aligned}$$

If m is an unambiguous parser, so is $m \rhd p$.

5.7 Iteration

A single parser may be iterated, yielding a list of parsed values.

$$\begin{aligned} &\mathit{iterate} &&:: M\ a \to M\ [a] \\ &\mathit{iterate}\ m = &&\quad (m \star \lambda a.\ \mathit{iterate}\ m \star \lambda x.\ \mathit{unit}\ (a : x)) \\ &&&\quad \oplus\ \mathit{unit}\ [\,] \end{aligned}$$

Given a parser m, the parser *iterate m* applies parser m in sequence zero or more times, returning a list of all the values parsed. In the list of alternative parses, the longest parse is returned first.

Here is an example.

$$\begin{aligned} \mathit{iterate}\ \mathit{digit}\ \text{“}\mathit{23\ \ and\ more}\text{”} = [&([\mathit{2}, \mathit{3}],\ \text{“}\ \mathit{and\ more}\text{”}), \\ &([\mathit{2}],\ \text{“}\mathit{3\ \ and\ more}\text{”}), \\ &([\,],\ \text{“}\mathit{23\ \ and\ more}\text{”})] \end{aligned}$$

Here is one way to parse a number.

$$\begin{aligned} &\mathit{number} :: M\ \mathit{Int} \\ &\mathit{number} = \mathit{digit} \star \lambda a.\ \mathit{iterate}\ \mathit{digit} \star \lambda x.\ \mathit{unit}\ (\mathit{asNumber}\ (a : x)) \end{aligned}$$

Here *asNumber* takes a list of one or more digits and returns the corresponding number. Here is an example.

$$\begin{aligned} \mathit{number}\ \text{“}\mathit{23\ \ and\ more}\text{”} = [&(\mathit{23},\ \text{“}\ \mathit{and\ more}\text{”}), \\ &(\mathit{2},\ \text{“}\mathit{3\ \ and\ more}\text{”})] \end{aligned}$$

This supplies two possible parses, one which parses both digits, and one which parses only a single digit. A number is defined to contain at least one digit, so there is no parse with zero digits.

As this last example shows, often it is more natural to design an iterator to yield only the longest possible parse. The next section describes a way to achieve this.

5.8 Biased choice

Alternation, written $m \oplus n$, yields all parses yielded by m followed by all parses yielded by n. For some purposes, it is more sensible to choose one or the other: all parses by m if there are any, and all parses by n otherwise. This is called biased choice.

$$\begin{aligned} &(\oslash) &&:: M\ a \to M\ a \to M\ a \\ &(m \oslash n)\ x = &&\textbf{if}\ \ m\ x \neq [\,]\ \ \textbf{then}\ \ m\ x\ \ \textbf{else}\ \ n\ x \end{aligned}$$

Biased choice, written $m \oslash n$, yields the same parses as m, unless m fails to yield any parse, in which case it yields the same parses as n.

Here is iteration, rewritten with biased choice.

$$\begin{aligned} &\mathit{reiterate} &&:: M\ a \to M\ [a] \\ &\mathit{reiterate}\ m = &&\quad (m \star \lambda a.\ \mathit{reiterate}\ m \star \lambda x.\ \mathit{unit}\ (a : x)) \\ &&&\quad \oslash\ \mathit{unit}\ [\,] \end{aligned}$$

The only difference is to replace $\oplus$ with $\oslash$. Instead of yielding a list of all possible parses with the longest first, this yields only the longest possible parse.

Here is the previous example revisited.

$$\mathit{reiterate}\ \mathit{digit}\ \text{“}23\ \mathit{and\ more}\text{”} = [([2, 3], \text{“}\ \mathit{and\ more}\text{”})]$$

In what follows, *number* is taken to be rewritten with *reiterate*.

$$\begin{aligned} &\mathit{number} :: M\ \mathit{Int} \\ &\mathit{number} = \mathit{digit} \star \lambda a.\ \mathit{reiterate}\ \mathit{digit} \star \lambda x.\ \mathit{unit}\ (\mathit{asNumber}\ (a : x)) \end{aligned}$$

Here is an example that reveals a little of how ambiguous parsers may be used to search a space of possibilities. We use *reiterate* to find all ways of taking one or two items from a string, zero or more times.

$$\begin{aligned} \mathit{reiterate}\ \mathit{oneOrTwoItems}\ \text{“}\mathit{many}\text{”} = [&([\text{“}m\text{”}, \text{“}a\text{”}, \text{“}n\text{”}, \text{“}y\text{”}], \text{“ ”}), \\ &([\text{“}m\text{”}, \text{“}a\text{”}, \text{“}ny\text{”}], \text{“ ”}), \\ &([\text{“}m\text{”}, \text{“}an\text{”}, \text{“}y\text{”}], \text{“ ”}), \\ &([\text{“}ma\text{”}, \text{“}n\text{”}, \text{“}y\text{”}], \text{“ ”}), \\ &([\text{“}ma\text{”}, \text{“}ny\text{”}], \text{“ ”})] \end{aligned}$$

This combines alternation (in *oneOrTwoItems*) with biased choice (in *reiterate*). There are several possible parses, but for each parse *oneOrTwoItems* has been applied until the entire input has been consumed. Although this example is somewhat fanciful, a similar technique could be used to find all ways of breaking a dollar into nickels, dimes, and quarters.

If m and n are unambiguous, then $m \oslash n$ and $\mathit{reiterate}\ m$ are also unambiguous. For unambiguous parsers, sequencing distributes right through biased choice:

$$(m \star k) \oslash (m \star h) = m \star \lambda a.\ k\ a \oslash h\ a$$

whenever m is unambiguous. Unlike with alternation, sequencing does not distribute left through biased choice, even for unambiguous parsers.

5.9 A parser for terms

It is now possible to write the parser for terms alluded to at the beginning. Here is a grammar for fully parenthesised terms, expressed in BNF.

$$\mathit{term} ::= \mathit{number} \mid \text{‘(’}\ \mathit{term}\ \text{‘÷’}\ \mathit{term}\ \text{‘)’}$$

This translates directly into our notation as follows. Note that our notation, unlike BNF, specifies exactly how to construct the returned value.

$$
\begin{array}{lll}
term :: M\ Term & & \\
term = & (number & \star\, \lambda a. \\
& \quad unit\,(Con\ a)) & \\
& \oplus\ (lit\ \text{`('} & \star\, \lambda\_. \\
& \quad term & \star\, \lambda t. \\
& \quad lit\ \text{`}\div\text{'} & \star\, \lambda\_. \\
& \quad term & \star\, \lambda u. \\
& \quad lit\ \text{`)'} & \star\, \lambda\_. \\
& \quad unit\,(Div\ t\ u)) &
\end{array}
$$

(Here $\lambda\_.\ e$ is equivalent to $\lambda x.\ e$ where x is some fresh variable that does not appear in e; it indicates that the value bound by the lambda expression is not of interest.) Examples of the use of this parser appeared earlier.

The above parser is written with alternation, but as it is unambiguous, it could just as well have been written with biased choice. The same is true for all the parsers in the next section.

5.10 Left recursion

The above parser works only for fully parenthesised terms. If we allow unparenthesised terms, then the operator $\div$ should associate to the left. The usual way to express such a grammar in BNF is as follows.

$$
\begin{array}{lll}
term & ::= & term\ \text{`}\div\text{'}\ factor \mid factor \\
factor & ::= & number \mid \text{`('}\ term\ \text{`)'}
\end{array}
$$

This translates into our notation as follows.

$$
\begin{array}{lll}
term & :: M\ Term & \\
term & = \quad (term & \star\, \lambda t. \\
& \quad lit\ \text{`}\div\text{'} & \star\, \lambda\_. \\
& \quad factor & \star\, \lambda u. \\
& \quad unit\,(Div\ t\ u)) & \\
& \oplus\ factor & \\
factor & :: M\ Term & \\
factor & = \quad (number & \star\, \lambda a. \\
& \quad unit\,(Con\ a)) & \\
& \oplus\ (lit\ \text{`('} & \star\, \lambda\_. \\
& \quad term & \star\, \lambda t. \\
& \quad lit\ \text{`)'} & \star\, \lambda\_. \\
& \quad unit\ t) &
\end{array}
$$

There is no problem with *factor*, but any attempt to apply *term* results in an infinite loop. The problem is that the first step of *term* is to apply *term*, leading

to an infinite regress. This is called the *left recursion problem*. It is a difficulty for all recursive descent parsers, functional or otherwise.

The solution is to rewrite the grammar for *term* in the following equivalent form.

$$\begin{aligned} term &::= factor\ term' \\ term' &::= \text{`}\div\text{'}\ factor\ term' \mid unit \end{aligned}$$

where as usual *unit* denotes the empty parser. This then translates directly into our notation.

$$\begin{aligned} term &:: M\ Term \\ term &= factor \star \lambda t.\ term'\ t \\ \\ term' &:: Term \to M\ Term \\ term'\ t &= (lit\ \text{`}\div\text{'} \quad \star \lambda\_. \\ &\qquad factor \quad \star \lambda u. \\ &\qquad term'\ (Div\ t\ u)) \\ &\quad \oplus\ unit\ t \end{aligned}$$

Here $term'$ parses the remainder of a term; it takes an argument corresponding to the term parsed so far.

This has the desired effect.

$$\begin{aligned} &term\ \text{“}1972 \div 2 \div 23\text{”} &&= [((Div\ (Div\ (Con\ 1972)\ (Con\ 2))\ (Con\ 23)),\ \text{“”})] \\ &term\ \text{“}1972 \div (2 \div 23)\text{”} &&= [((Div\ (Con\ 1972)\ (Div\ (Con\ 2)\ (Con\ 23))),\ \text{“”})] \end{aligned}$$

In general, the left-recursive definition

$$m = (m \star k) \oplus n$$

can be rewritten as

$$m = n \star (closure\ k)$$

where

$$\begin{aligned} closure &:: (a \to M\ a) \to (a \to M\ a) \\ closure\ k\ a &= (k\ a \star closure\ k) \oplus unit\ a \end{aligned}$$

Here $m :: M\ a$, $n :: M\ a$, and $k :: a \to M\ a$.

5.11 Improving laziness

Typically, a program might be represented as a function from a list of characters – the input – to another list of characters – the output. Under lazy evaluation, usually only some of the input need be read before the first part of the output list is produced. This ‘on line’ behavior is essential for some purposes.

In general, it is unreasonable to expect such behaviour from a parser, since in general it cannot be known that the input will be successfully parsed until all of it is read. However, in certain special cases one may hope to do better.

Consider applying *reiterate m* to a string beginning with an instance of m. In this case, the parse cannot fail: regardless of the remainder of the string, one would expect the parse yielded to be a list beginning with the parsed value. Under

lazy evaluation, one might expect to be able to generate output corresponding to the first digit before the remaining input has been read.

But this is not what happens: the parser reads the entire input before any output is generated. What is necessary is some way to encode that the parser *reiterate m* always succeeds. (Even if the beginning of the input does not match m, it will yield as a value the empty list.) This is provided by the function *guarantee*.

$$\begin{aligned} &\mathit{guarantee} &&:: M\ a \to M\ a \\ &\mathit{guarantee}\ m\ x = \mathbf{let}\ \ u = m\ x\ \ \mathbf{in}\ \ (\mathit{fst}\ (\mathit{head}\ u), \mathit{snd}\ (\mathit{head}\ u)) : \mathit{tail}\ u \end{aligned}$$

Here $\mathit{fst}\ (a, b) = a$, $\mathit{snd}\ (a, b) = b$, $\mathit{head}\ (a : x) = a$, and $\mathit{tail}\ (a : x) = x$.

Here is *reiterate* with the guarantee added.

$$\begin{aligned} &\mathit{reiterate} &&:: M\ a \to M\ [a] \\ &\mathit{reiterate}\ m = \mathit{guarantee}\ (\ \ (m \star \lambda a.\ \mathit{reiterate}\ m \star \lambda x.\ \mathit{unit}\ (a : x)) \\ &\qquad\qquad\qquad\qquad \oslash\ \mathit{unit}\ [\,]) \end{aligned}$$

This ensures that *reiterate m* and all of its recursive calls return a list with at least one answer. As a result, the behaviour under lazy evaluation is much improved.

The preceding explanation is highly operational, and it is worth noting that denotational semantics provides a useful alternative approach. Let $\bot$ denote a program that does not terminate. One can verify that with the old definition

$$\mathit{reiterate}\ \mathit{digit}\ (\textit{'1'} : \bot) = \bot$$

while with the new definition

$$\mathit{reiterate}\ \mathit{digit}\ (\textit{'1'} : \bot) = ((\textit{'1'} : \bot), \bot) : \bot$$

Thus, given that the input begins with the character *'1'* but that the remainder of the input is unknown, with the old definition nothing is known about the output, while with the new definition it is known that the output yields at least one parse, the value of which is a list which begins with the character *'1'*.

Other parsers can also benefit from a judicious use of *guarantee*, and in particular *iterate* can be modified like *reiterate*.

5.12 Conclusion

We have seen that monads provide a useful framework for structuring recursive descent parsers. The empty parser and sequencing correspond directly to *unit* and $\star$, and the monads laws reflect that sequencing is associative and has the empty parser as a unit. The failing parser and alternation correspond to *zero* and $\oplus$, which satisfy laws reflecting that alternation is associative and has the failing parser as a unit, and that sequencing distributes through alternation.

Sequencing and alternation are fundamental not just to parsers but to much of computing. If monads capture sequencing, then it is reasonable to ask: what

captures both sequencing and alternation? It may be that *unit*, $\star$, *zero*, and $\oplus$, together with the laws above, provide such a structure. Further experiments are needed. One hopeful indication is that a slight variation of the parser monad yields a plausible model of Dijkstra's guarded command language.

References

1. S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
2. A. Bloss, Update analysis and the efficient implementation of functional aggregates. In *4'th Symposium on Functional Programming Languages and Computer Architecture*, ACM, London, September 1989.
3. R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall, 1987.
4. P. Hudak, S. Peyton Jones and P. Wadler, editors, *Report on the Programming Language Haskell: Version 1.1*. Technical report, Yale University and Glasgow University, August 1991.
5. J.-Y. Girard, Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
6. J. Guzmán and P. Hudak, Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, Philadelphia, June 1990.
7. P. Hudak, A semantic model of reference counting and its abstraction (detailed summary). In *ACM Conference on Lisp and Functional Programming*, pp. 351–363, Cambridge, Massachusetts, August 1986.
8. P. Hudak, Continuation-based mutable abstract data types, or how to have your state and munge it too. Technical report YALEU/DCS/RR-914, Department of Computer Science, Yale University, July 1992.
9. D. King and P. Wadler, Combining monads. In *Glasgow Workshop on Functional Programming*, Ayr, July 1992. Workshops in Computing Series, Springer Verlag, to appear.
10. S. Mac Lane, *Categories for the Working Mathematician*, Springer-Verlag, 1971.
11. R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML*. MIT Press, 1990.
12. L. C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1991.
13. E. Moggi, Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar, California; IEEE, June 1989. (A longer version is available as a technical report from the University of Edinburgh.)
14. E. Moggi, An abstract view of programming languges. Course notes, University of Edinburgh.
15. S. L. Peyton Jones and P. Wadler, Imperative functional programming. In *20'th Symposium on Principles of Programming Languages*, Charleston, South Carolina; ACM, January 1993.
16. G. Plotkin, Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
17. J. Rees and W. Clinger (eds.), The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.
18. D. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, 7:299–310, 1985.

19. V. Swarup, U. S. Reddy, and E. Ireland, Assignments for applicative languages. In *Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusetts; LNCS 523, Springer Verlag, August 1991.
20. D. A. Turner, An overview of Miranda. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.
21. P. Wadler, Comprehending monads. In *Conference on Lisp and Functional Programming*, Nice, France; ACM, June 1990.
22. Is there a use for linear logic? *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, New Haven, Connecticut; ACM, June 1991.
23. P. Wadler, The essence of functional programming (invited talk). In *19'th Symposium on Principles of Programming Languages*, Albuquerque, New Mexico; ACM, January 1992.

The Haskell Programmer's Guide to the IO Monad

— Don't Panic —

Stefan Klinger

University of Twente, the Netherlands · EWI, Database Group
CTIT Technical Report

Stefan Klinger.
The Haskell Programmer's Guide to the IO Monad — Don't Panic.

Order-address:
Centre for Telematics and Information Technology
University of Twente
P. O. Box 217
7500 AL Enschede
the Netherlands
mailto:a.m.annink-tanke@utwente.nl

PDF available at http://stefan-klinger.de.

Stefan Klinger. The Haskell Programmer's Guide to the IO Monad — Don't Panic. Technical report, December 2005, no. 05-54, 33 pp., Centre for Telematics and Information Technology (CTIT), ISSN 1381-3625.

Version of 2005-Dec-15 15:39:54 .

Preface

Now, that you have started with Haskell, have you written a program doing IO yet, like reading a file or writing on the terminal? Then you have used the *IO monad* — but do you understand how it works?

The standard explanation is, that the IO monad hides the non-functional *IO actions* —which do have side effects— from the functional world of Haskell. It prevents pollution of the functional programming style with side effects.

However, since most beginning Haskell programmers (i.e., everyone I know and including me) lack knowledge about category theory, they have no clue about what a monad really is. Nor how this "hiding" works, apart from having IO actions disappearing beyond the borders of our knowledge.

This report scratches the surface of category theory, an abstract branch of algebra, just deep enough to find the monad structure. On the way we discuss the relations to the purely functional programming language Haskell. Finally it should become clear how the IO monad keeps Haskell pure.

We do not explain how to use the IO monad, nor discuss all the functions available to the programmer. But we do talk about the theory behind it.

Intended audience Haskell programmers that stumbled across the IO monad, and now want to look under the hood. Haskell experience and the ability to read math formulae are mandatory.

Many thanks to Sander Evers, Maarten Fokkinga, and Maurice van Keulen for reviewing, a lot of discussions, helpful insights and suggestions.

Contents

Chapter 1

Introduction

1.1 Motivation

Imagine you want to write a somewhat more sophisticated "hello world" program in Haskell, which asks for the user's name and then addresses him personally.

You probably end up with something like

```
main :: IO ()
main = do putStr "What's your name?\n> "
          x <- getLine
          putStr . concat $ [ "Hello ", x, ".\n" ]
```

So what exactly does the "`do`" mean? Is "`<-`" variable assignment? And what does the type "`IO ()`" mean?

Having accepted the presence of "`IO ()`" in the type signatures of basic IO functions (like, e.g., `putStr`, `getChar`), a lot of novice Haskell programmers try to get rid of it as soon as possible:

> *"I'll just wrap the* `getLine` *in another function which only returns what I really need, maybe convert the users' input to an* `Int` *and return that."*

However, this is not possible since it would violate the *referential transparency* Haskell enforces for all functions. Referential transparency guarantees that a function, given the same parameters, always returns the same result.

How can IO be done at all, if a function is referential transparent and must not have side effects? Try to write down the signatures of such a function that reads a character from the keyboard, and a function that writes a character to the screen, both without using the IO monad.

The intention of this guide is to show how the Monad helps in doing IO in a purely functional programming language, by illuminating the mathematical background of this structure.

1.2 Related work

This guide tries to fit exactly between the available theoretical literature about category theory on the one side (e.g., [1], [2]) and literature about how to program with monads ([3], [4]) on the other.

However, the sheer amount of available literature on both sides of this report dooms any approach to offer a complete list of related work to failure.

Chapter 2

Notation

I tried to keep the notation as readable as possible, yet unambiguous. The meaning should be clear immediately, and I hope that predicate logic people excuse some lack of purity.

1. Quite often proofs are interspersed with explanatory text, e.g., talking about integers we might note

   $$\begin{array}{ll} & a+b \\ = & \quad [\text{ operator '}+\text{' is commutative} \\ & b+a \ . \end{array}$$

2. Function application is always noted in juxtaposition —i.e., the operand is just written behind the function— to avoid a Lisp-like amount of parenthesis, i.e.

   $$\begin{array}{ll} & fx \\ \equiv & \quad [\text{ by definition} \\ & f(x) \ . \end{array}$$

3. Quantifiers have higher precedence than *and* and *or* (symbols $\wedge, \vee$), but lower than *implication* and *equivalence* ($\Rightarrow, \Leftrightarrow$).

   A quantifier "binds" all the free variables in the following ';'-separated list of predicates that are neither given in the context of the formula (constants), nor bound by an earlier quantifier. I.e.,

   $$\begin{array}{ll} & \forall\, m,n \in \mathbb{N}\,;\ m<n \quad \exists\, r \in \mathbb{R} \quad m<r<n \\ \equiv & \quad [\text{ next line is predicate logic} \\ & \forall m \forall n \big(m \in \mathbb{N} \wedge n \in \mathbb{N} \wedge m<n \ \Rightarrow\ \exists r (r \in \mathbb{R} \wedge m<r<n) \big) \ . \end{array}$$

4. Sometimes the exercise sign (✎) occurs, requesting the reader to verify something, or to play with the Haskell code given.

5. A λ-expression binding the free occurrences of y in expression e is written

   $$(y \mapsto e) \ .$$

   As a Haskell programmer, you should be familiar with λ-expressions.

6. Unless stated otherwise, most programming code is Haskell-*pseudo*code. In contradiction to this, Haskell-code that should be tried out by the user is referred to with a ✎-sign.

Chapter 3

Categories

3.1 Categories in theory

Introductory example Let us start with a quite concrete example of a category: Sets (later called objects) together with all the total functions on them (called morphisms):

- It is common practice to write $f : A \to B$ to denote that a function f maps from set A to set B. This is called the *type* of the function.
- Also, we can compose two functions f and g, if the target set of the former equals the source set of the latter, i.e., if $f : A \to B$ and $g : B \to C$ for some sets A, B, and C. The composition is commonly denoted by $g \circ f$.
- For each set A, there is an identity function $\mathrm{id}_A : A \to A$.

These are the properties of a category. While most things in this guide can be applied to the category of sets, Definition 3.1.1 is more precise and general. In particular, category theory is not restricted to sets and total functions, i.e., one can not assume that an object has elements (like sets do), or that an element is mapped to another element.

3.1.1 Definition A **category** $\mathcal{C} = (\mathcal{O}_\mathcal{C}, \mathcal{M}_\mathcal{C}, \mathcal{T}_\mathcal{C}, \circ_\mathcal{C}, \mathrm{Id}_\mathcal{C})$ is a structure consisting of *morphisms* $\mathcal{M}_\mathcal{C}$, *objects* $\mathcal{O}_\mathcal{C}$, a *composition* $\circ_\mathcal{C}$ of morphisms, and *type information* $\mathcal{T}_\mathcal{C}$ of the morphisms, which obey to the following constraints.

Note, that we often omit the subscription with $\mathcal{C}$ if the category is clear from the context.

1. We assume the collection of **objects** $\mathcal{O}$ to be a set. Quite often the objects themselves also are sets, however, category theory makes *no* assumption about that, and provides no means to explore their structure.
2. The collection of **morphisms** $\mathcal{M}$ is also assumed to be a set in this guide.
3. The ternary relation $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{O} \times \mathcal{O}$ is called the **type information** of $\mathcal{C}$. We want every morphism to have a type, so a category requires

   $$\forall\, f \in \mathcal{M} \quad \exists\, A, B \in \mathcal{O} \quad (f, A, B) \in \mathcal{T} \ .$$

   We write $f : A \xrightarrow[\mathcal{C}]{} B$ for $(f, A, B) \in \mathcal{T}_\mathcal{C}$. Also, we want the types to be unique, leading to the claim

   $$f : A \to B \wedge f : A' \to B' \ \Rightarrow\ A = A' \wedge B = B' \ .$$

   This gives a notion of having a morphism to "map from one object to another". The uniqueness entitles us to give names to the objects involved in a morphism. For $f : A \to B$ we call $\mathrm{src}\, f := A$ the **source**, and $\mathrm{tgt}\, f := B$ the **target** of f.
4. The *partial* function $\circ : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, written as a binary infix operator, is called the **composition** of $\mathcal{C}$. Alternative notations are $f ; g := gf := g \circ f$.

Morphisms can be composed whenever the target of the first equals the source of the second. Then the resulting morphism maps from the source of the first to the target of the second:

$$f : A \to B \wedge g : B \to C \;\Rightarrow\; g \circ f : A \to C$$

In the following, the notation $g \circ f$ implies these type constraints to be fulfilled.

Composition is associative, i.e. $f \circ (g \circ h) = (f \circ g) \circ h$.

5. Each object $A \in \mathcal{O}$ has associated a unique **identity morphism** id_A. This is denoted by defining the function

$$\begin{array}{rccc} \mathrm{Id} & : & \mathcal{O} & \longrightarrow & \mathcal{M} \\ & & A & \longmapsto & \mathrm{id}_A \end{array} ,$$

which automatically implies uniqueness.

The typing of the identity morphisms adheres to

$$\forall\, A \in \mathcal{O} \quad \mathrm{id}_A : A \to A \ .$$

To deserve their name, the identity morphisms are —depending on the types— left or right neutral with respect to composition, i.e.,

$$f \circ \mathrm{id}_{\mathrm{src}\, f} = f = \mathrm{id}_{\mathrm{tgt}\, f} \circ f \ .$$

Another example Every set A with a partial order $(\leq)$ on it yields a category, say $\mathcal{Q}$. To see this, call the elements of A objects, and the relation morphisms:

$$\mathcal{O}_\mathcal{Q} := A \ ,$$
$$\mathcal{M}_\mathcal{Q} := \{(x, y) \mid x, y \in A \wedge x \leq y\} \ .$$

Now (✎ as an exercise) define the type information $\mathcal{T}_\mathcal{Q}$, the composition $\circ_\mathcal{Q}$ and the identities $\mathrm{Id}_\mathcal{Q}$, so that $\mathcal{Q}$ is a category indeed. Note, that this example does not assume the objects to have any members.

More examples and a broader introduction to category theory (however, without discussing the monad structure) can be found in [1].

3.2 Spot a category in Haskell

This guide looks at one particular category that can be recognised in the Haskell programming language. There might be others of more or less interest, but for the purpose of explaining Haskell's Monad structure this narrow perspective is sufficient. For a more thorough discussion about functional programming languages and category theory you might read [2].

With the last section in mind, where would you look for "the obvious" category? It is not required that it models the whole Haskell language, instead it is enough to point at the things in Haskell that behave like the objects and morphisms of a category.

We call our Haskell category $\mathcal{H}$ and use Haskell's types —primitive as well as constructed— as the objects $\mathcal{O}_\mathcal{H}$ of the category. Then, unary Haskell functions correspond to the morphisms $\mathcal{M}_\mathcal{H}$, with function signatures of unary functions corresponding to the type information $\mathcal{T}_\mathcal{H}$.

`f :: A -> B` corresponds to $f : A \underset{\mathcal{H}}{\longrightarrow} B$

Haskell's function composition '`.`' corresponds to the composition of morphisms $\circ_\mathcal{H}$. The identity in Haskell is typed

```
id :: forall a. a -> a
```

corresponding to

$$\forall\, A \in \mathcal{O}_\mathcal{H} \quad \mathrm{id}_A : A \to A \ .$$

Note that we do not talk about n-ary functions for $n \neq 1$. You can consider a function like $(+)$ to map a number to a function that adds this number, a technique called **currying** which is widely used by Haskell programmers. Within the context of this guide, we do not treat the resulting function as a morphism, but as an object.

Chapter 4

Functors

4.1 Functors in theory

One can define mappings between categories. If they "behave well", i.e., preserve the structural properties of being an object, morphism, identity, the types and composition, they are called *functors*.

4.1.1 Definition Let $\mathcal{A}, \mathcal{B}$ be categories. Then two mappings

$$F_{\mathcal{O}} : \mathcal{O}_{\mathcal{A}} \to \mathcal{O}_{\mathcal{B}} \quad \text{and} \quad F_{\mathcal{M}} : \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{B}}$$

together form a **functor** F from $\mathcal{A}$ to $\mathcal{B}$, written $F : \mathcal{A} \to \mathcal{B}$, iff

1. they preserve type information, i.e.,
$$\forall\, f : A \underset{\mathcal{A}}{\longrightarrow} B \quad F_{\mathcal{M}} f : F_{\mathcal{O}} A \underset{\mathcal{B}}{\longrightarrow} F_{\mathcal{O}} B \ ,$$
2. $F_{\mathcal{M}}$ maps identities to identities, i.e.,
$$\forall\, A \in \mathcal{O}_{\mathcal{A}} \quad F_{\mathcal{M}} \operatorname{id}_A = \operatorname{id}_{F_{\mathcal{O}} A} \ ,$$
3. and application of $F_{\mathcal{M}}$ distributes under composition of morphisms, i.e.,
$$\forall\, f : A \underset{\mathcal{A}}{\longrightarrow} B\,;\ g : B \underset{\mathcal{A}}{\longrightarrow} C \quad F_{\mathcal{M}}(g \circ_{\mathcal{A}} f) = F_{\mathcal{M}} g \circ_{\mathcal{B}} F_{\mathcal{M}} f \ .$$

4.1.2 Notation Unless it is required to refer to only one of the mappings, the subscripts $_{\mathcal{M}}$ and $_{\mathcal{O}}$ are usually omitted. It is clear from the context whether an object or a morphism is mapped by the functor.

4.1.3 Definition Let $\mathcal{A}$ be a category. A functor $F : \mathcal{A} \to \mathcal{A}$ is called **endofunctor**.

It is easy to define a "identity" on a category, by simply combining the identities on objects and morphisms the same way we just have combined the functors. Also, applying one functor after another —where the target category of the first must be the source category of the second— looks like composition:

4.1.4 Definition Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be categories, $F : \mathcal{A} \to \mathcal{B}$ and $G : \mathcal{B} \to \mathcal{C}$. We define **identities**

$$I_{\mathcal{A}} := \operatorname{id}_{\mathcal{O}_{\mathcal{A}} \uplus \mathcal{M}_{\mathcal{A}}}$$

and **functor composition** GF with

$$\forall\, A \in \mathcal{O} \quad (GF)A := G(FA)$$
$$\forall\, f \in \mathcal{M} \quad (GF)f := G(Ff) \ .$$

Due to this definition, we can write GFA and GFf without ambiguity. Multiple application of the same mapping is often noted with a superscript, i.e., we define $F^2 := FF$ for any functor F.

4.1.5 Lemma The identity and composition just defined yield functors again: In the situation of Definition 4.1.4,

$$I_{\mathcal{A}} : \mathcal{A} \to \mathcal{A} \ , \qquad GF : \mathcal{A} \to \mathcal{C} \ , \qquad I_{\mathcal{B}} F = F = F I_{\mathcal{A}} \ .$$

4.1.6 Proof is easy (✎). □

With that in mind, one can even consider categories to be the objects, and functors to be the morphisms of some higher level category (a category of categories). Although we do not follow this path, we do use composition of functors and identity functors in the sequel. However, we use the suggestive notation GF instead of $G \circ F$ since we do not discuss category theory by its own means.

4.2 Functors implement structure

The objects of a category may not have any structure, the structure may not be known to us, or the structure may not be suitable for our programming task, just as characters without additional structure are not suitable to represent a string. Functors are the tool to add some structure to an object.

A functor L may implement the List structure by mapping the object "integers" to the object "list of integers", and a function f that operates on integers, to a function Lf which applies f to each element of the list, returning the list of results. The same —suitably defined— functor L would map any object in the category to the corresponding list object, like the object "characters" to the object "strings", and booleans to bit-vectors.

In this example, a (not *the*) suitable category $\mathcal{S}$ for $L : \mathcal{S} \to \mathcal{S}$ would be the category where the objects are all sets, and the morphisms are all total functions between sets — that is our introductory example. Note, that the structure of the integers themselves is not affected by the functor application, hence we can say that the structure is added *on the outside* of the object. The addition of structure by application of a functor is often referred to using the term **lifting**, like in "*the integers are lifted to lists of integers*" or "*the lifted function now operates on lists.*"

This vague explanation of how a functor describes a structure is refined during the remainder of this guide. But already at this point, you can try to think about endofunctors that manifest structures like pairs, n-tuples, sets, and so on (✎ define some of these). Obviously, functor composition implements the nesting of structures, leading to structures like "list of pairs", "pair of lists", etc.

4.3 Functors in Haskell

Following our idea of a Haskell category $\mathcal{H}$, we can define an endofunctor $F : \mathcal{H} \to \mathcal{H}$ by giving a unary type constructor `F`, and a function `fmap`, constituting $F_{\mathcal{O}}$ and $F_{\mathcal{M}}$ respectively.

The type constructor `F` is used to construct new types from existing ones. This action corresponds to mapping objects to objects in the $\mathcal{H}$ category. The definition of `F` shows how a functor implements the structure of the constructed type. The function `fmap` lifts each function with a signature $f : A \to B$ to a function with signature $Ff : FA \to FB$.

Hence, functor application on an object is a *type level* operation in Haskell, while functor application on a morphism is a *value level* operation.

Haskell comes with the definition of a class

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

which allows overloading of `fmap` for each functor.

✎ Working examples are the *List* or *Maybe* structures (see Lemma 4.3.1). Both are members of the functor class, so you can enter the following lines into your interactive Haskell interpreter. The `:t` prefix causes *Hugs* and *GHCi* to print the type information of the following expression. This might not work in every environment. Understand the results returned by the Haskell interpreter.

```
--loading 'Char' seems to be required in GHCi, but not in Hugs
:m Char

:t fmap ord
fmap ord ""
fmap ord "lambda"
:t fmap chr
fmap chr Nothing
fmap chr (Just 42)
```

The examples illustrate how the functions `ord` and `chr` (from Haskell's Char module) are lifted to work on Lists and Maybes instead of just atomic values.

> You might stumble across the question what data constructors like `Just`, `Nothing`, `(:)` and `[]` actually *are*. At this point we are leaving category theory, and look *into* the objects which indeed turn out to have set properties in Haskell.
>
> A sound discussion of Haskell's type system is far beyond the scope of this little guide, so just imagine the data constructors to be some **markers** or **tags** to describe a member of an object. I.e., `Just "foo"` describes a member of the `Maybe String` object, as does the polymorphic `Nothing`.

The mapping function `fmap` differs from functor to functor. For the Maybe structure, it can be written as

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

while `fmap` for the List structure can be written as

```
fmap :: (a -> b) -> [a] -> [b]
fmap f [] = []
fmap f (x:xs) = (f x):(fmap f xs)
```

Note, however, that having a type being a member of the `Functor` class does not imply to have a functor at all. Haskell does not check validity of the functor properties, so we have to do it by hand:

1. $f : A \to B \;\Rightarrow\; Ff : FA \to FB$ is fulfilled, since being a member of the Functor class implies that a function `fmap` with the according signature is defined.
2. $\forall\ A \in \mathcal{O} \quad F\,\mathrm{id}_A = \mathrm{id}_{FA}$ translates to

   ```
   fmap id == id
   ```

   which must be checked for each type one adds to the class. Note, that Haskell overloads the `id` function: The left occurrence corresponds to id_A, while the right one corresponds to id_{FA}.
3. $F(g \circ f) = Fg \circ Ff$ translates to

   ```
   fmap (g . f) == fmap g . fmap f
   ```

   which has to be checked, generalising over all functions `g` and `f` of appropriate type.

One should never add a type constructor to the `Functor` class that does not obey these laws, since this would be misleading for people reading the produced code.

4.3.1 Lemma Maybe and List are both functors.

I.e., they are not only members of the class, but also behave as expected in theory. The proof is easy (✎), and when ever you want to add a type to the functor class, you have to perform this kind of proof. So *do this as an exercise* before reading ahead.

4.3.2 Proof The proof of Lemma 4.3.1 strictly follows the structure of the Haskell types, i.e., we differentiate according to the data constructors used.

For the Maybe structure, the data constructors are `Just` and `Nothing`. So we prove the second functor property, $F\,\mathrm{id}_A = \mathrm{id}_{FA}$, by

```
fmap id Nothing
  == Nothing
  == id Nothing

fmap id (Just y)
  == Just (id y)
  == Just y
  == id (Just y) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$, by

```
(fmap g . fmap h) Nothing
  == fmap g (fmap h Nothing)
  == fmap g Nothing
  == Nothing
  == fmap (g . h) Nothing ,

(fmap g . fmap h) (Just y)
  == fmap g (fmap h (Just y))
  == fmap g (Just (h y))
  == Just (g (h y))
  == Just ((g . h) y)
  == fmap (g . h) (Just y)  .
```

Note, that List is —in contrast to Maybe— a recursive structure. This can be observed in the according definition of `fmap` above, and it urges us to use induction in the proof: The second property, $F\,\mathrm{id}_A = \mathrm{id}_{FA}$, is shown by

```
fmap id []
  == []
  == id []

fmap id (x:xs)
  == (id x):(fmap id xs)
  == (id x):(id xs)  --here we use induction
  == x:xs
  == id (x:xs) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$ can be observed in

```
fmap (g . f) []
  == []
  == fmap g []
  == fmap g (fmap f [])
  == (fmap g . fmap f) []

fmap (g . f) (x:xs)
  == ((g . f) x):(fmap (g . f) xs)
  == ((g . f) x):((fmap g . fmap f) xs)  --induction
  == (g (f x)):(fmap g (fmap f xs))
  == fmap g ((f x):(fmap f xs))
  == fmap g (fmap f xs)
  == (fmap g . fmap f) xs  .
```

□

Chapter 5

Natural Transformations

5.1 Natural transformations in theory

A natural transformation is intended to transform from one structure to another, without effecting, or being influenced by, the objects in the structure.

Imagine two functors F and G —imposing two structures— between the same categories $\mathcal{A}$ and $\mathcal{B}$. Then a natural transformation is a collection of morphisms in the target category $\mathcal{B}$ of both functors, always mapping from an object FA to an object GA. In fact, for each object A in the source category there is one such morphism in the target category:

5.1.1 Definition Let $\mathcal{A}, \mathcal{B}$ be categories, $F, G : \mathcal{A} \to \mathcal{B}$. Then, a function

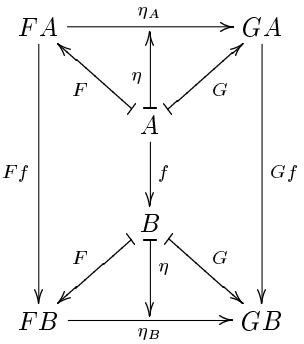
$$\begin{array}{rccc} \eta & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\ & & A & \longmapsto & \eta_A \end{array}$$

is called a **transformation** from F to G, iff

$$\forall\, A \in \mathcal{O}_{\mathcal{A}} \quad \eta_A : FA \underset{\mathcal{B}}{\longrightarrow} GA \ ,$$

and it is called a **natural** transformation, denoted $\eta : F \dot{\to} G$, iff

$$\forall\, f : A \underset{\mathcal{A}}{\longrightarrow} B \quad \eta_B \circ_{\mathcal{B}} Ff = Gf \circ_{\mathcal{B}} \eta_A \ .$$



The definition says, that a natural transformation *transforms* from a structure F to a structure G, without altering the behaviour of morphisms on objects. I.e., it does not play a role whether a morphism f is lifted into the one or the other structure, when composed with the transformation.
In the drawing on the left, this means that the outer square *commutes*, i.e., all directed paths with common source and target are equal.

5.2 Natural transformations in Haskell

First note, that a unary function polymorphic in the same type on source and target side, in fact is a transformation. For example, Haskell's `Just` is typed

```
Just :: forall a. a -> Maybe a  .
```

If we imagine this to be a mapping $\texttt{Just} : \mathcal{O}_{\mathcal{H}} \to \mathcal{M}_{\mathcal{H}}$, the application on an object $A \in \mathcal{O}_{\mathcal{H}}$ means binding the type variable `a` to some Haskell type `A`. That is, `Just` maps an object A to a morphism of type $A \to \texttt{Maybe}\ A$.

To spot a transformation here, we still miss a functor on the source side of `Just`'s type. This is overcome by adding the identity functor $I_{\mathcal{H}}$, which leads to the transformation

$$\texttt{Just} : I_{\mathcal{H}} \to \texttt{Maybe} \ .$$

5.2.1 Lemma $\texttt{Just} : I_{\mathcal{H}} \dot{\to} \texttt{Maybe}$.

5.2.2 Proof Let `f::A->B` be an arbitrary Haskell function. Then we have to prove, that

```
Just . I_H f == fmap f . Just
```

where the left `Just` refers to η_B, and the right one refers to η_A. In that line, we recognise the definition of the `fmap` for Maybe (just drop the $I_{\mathcal{H}}$). □

Another example, this time employing two non-trivial functors, are the `maybeToList` and `listToMaybe` functions. Their definitions read

```
maybeToList :: forall a. Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just x) = [x]

listToMaybe :: forall a. [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (x:xs) = Just x .
```

Note, that the *loss of information* imposed by applying `listToMaybe` on a list with more than one element does not contradict naturality, i.e., *forgetting* data is not *altering* data.

We do not go through the proof of their naturality here (✎ but, of course, you can do this on your own). Instead, we discuss some more interesting examples in the next chapter (see Lemma 6.2.1).

5.3 Composing transformations and functors

To describe the Monad structure later on, we need to compose natural transformations with functors and with other natural transformations. This is discussed in the remainder of this chapter.

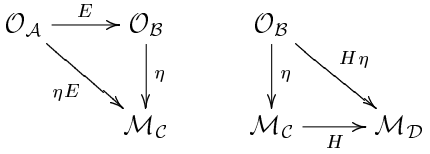
Natural transformations and functors Just as in Definition 4.1.4, we can define a composition between a natural transformation and a functor. Therefore, an object is first lifted by the functor and then mapped to a morphism by the transformation, or it is first mapped to a morphism which is then lifted:

5.3.1 Definition Let $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ be categories, $E : \mathcal{A} \to \mathcal{B}$, $F, G : \mathcal{B} \to \mathcal{C}$, and $H : \mathcal{C} \to \mathcal{D}$. Then, for a natural transformation $\eta : F \dot{\to} G$, we define the transformations ηE and $H\eta$ with

$$(\eta E)\, A := \eta_{EA} \quad \text{and} \quad (H\eta) B := H(\eta_B) \ ,$$

where A varies over $\mathcal{O}_{\mathcal{A}}$, and B varies over $\mathcal{O}_{\mathcal{B}}$.

Again, due to the above definition, we can write ηEA and $H\eta B$ (for A and B objects in the respective category) without ambiguity. The following pictures show the situation:



5.3.2 Lemma In the situation of Definition 5.3.1,

$$H\eta : HF \dot{\to} HG \quad \text{and} \quad \eta E : FE \dot{\to} GE$$

hold. In (other) words, $H\eta$ and ηE are both natural transformations.

5.3.3 Proof of Lemma 5.3.2

▷ **Part I**

$$
\begin{array}{ll}
 & \eta : F \dot{\longrightarrow} G \ , \\
\Rightarrow & \quad [\text{ definition of naturality} \\
 & \forall\ f : A \underset{\mathcal{B}}{\longrightarrow} B \quad \eta B \circ Ff = Gf \circ \eta A \\
\Rightarrow & \forall\ f : A \underset{\mathcal{B}}{\longrightarrow} B \quad H(\eta B \circ Ff) = H(Gf \circ \eta A) \\
\Rightarrow & \quad [\text{ functor property} \\
 & \forall\ f : A \underset{\mathcal{B}}{\longrightarrow} B \quad H(\eta B) \circ H(Ff) = H(Gf) \circ H(\eta A) \\
\Rightarrow & \quad [\text{ Definition 5.3.1} \\
 & \forall\ f : A \underset{\mathcal{B}}{\longrightarrow} B \quad (H\eta)B \circ (HF)f = (HG)f \circ (H\eta)A \\
\Rightarrow & \quad [\text{ definition of naturality} \\
 & H\eta : HF \dot{\longrightarrow} HG \ ,
\end{array}
$$

▷ **Part II**

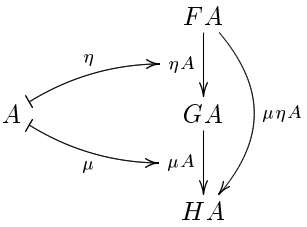
$$
\begin{array}{ll}
 & \eta : F \dot{\longrightarrow} G \\
\Rightarrow & \quad [\text{ definition of naturality} \\
 & \forall\ f : A \underset{\mathcal{B}}{\longrightarrow} B \quad \eta B \circ Ff = Gf \circ \eta A \\
\Rightarrow & \quad [\text{ choose } f \text{ from category } \mathcal{A} \\
 & \forall\ f : A \underset{\mathcal{A}}{\longrightarrow} B \quad \eta(EB) \circ F(Ef) = G(Ef) \circ \eta(EA) \\
\Rightarrow & \quad [\text{ Definition 5.3.1} \\
 & \forall\ f : A \underset{\mathcal{A}}{\longrightarrow} B \quad (\eta E)B \circ (FE)f = (GE)f \circ (\eta E)A \\
\Rightarrow & \quad [\text{ definition of naturality} \\
 & \eta E : FE \dot{\longrightarrow} GH \ .
\end{array}
$$

□

Composing natural transformations Even natural transformations can be composed with each other. Since, however, transformations map objects to morphisms —instead of, e.g., objects to objects— they can not be simply applied one after another. Instead, composition is defined *component wise*, also called **vertical composition**.

5.3.4 Definition Let $\mathcal{A}, \mathcal{B}$ be categories, $F, G, H : \mathcal{A} \to \mathcal{B}$ and $\eta : F \dot{\longrightarrow} G$, $\mu : G \dot{\longrightarrow} H$. Then we define the transformations

$$
\begin{array}{rcccl}
\mathrm{id}_F & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\
 & & A & \longmapsto & \mathrm{id}_{FA}
\end{array}
\quad \text{and} \quad
\begin{array}{rcccl}
\mu\eta & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\
 & & A & \longmapsto & \mu A \circ \eta A \ .
\end{array}
$$



The picture on the left (with $A \in \mathcal{O}_{\mathcal{A}}$) clarifies why this composition is called *vertical*.
All compositions defined before Definition 5.3.4 are called **horizontal**. ✎ Why?

5.3.5 Lemma In the situation of Definition 5.3.4,

$$\begin{aligned}
&\mathrm{id}_F : F \dot{\longrightarrow} F \ , \\
&\mu\eta : F \dot{\longrightarrow} H \ , \\
&\mathrm{id}_G \, \eta = \eta = \eta \, \mathrm{id}_F
\end{aligned}$$

hold.

5.3.6 Proof of Lemma 5.3.5.
The naturality of id_F is trivial (✎). For the naturality of $\mu\eta$ consider any morphism $f : A \underset{\mathcal{A}}{\longrightarrow} B$. Then,

$$\begin{aligned}
&\mu : G \dot{\longrightarrow} H \quad \wedge \quad \eta : F \dot{\longrightarrow} G \\
\Rightarrow\ &Hf \circ \mu A = \mu B \circ Gf \quad \wedge \quad Gf \circ \eta A = \eta B \circ Ff \\
\Rightarrow\ &\quad \big[\text{ apply } \eta A \text{ on the left, } \mu B \text{ on the right} \\
&(Hf \circ \mu A) \circ \eta A = (\mu B \circ Gf) \circ \eta A \quad \wedge \quad \mu B \circ (Gf \circ \eta A) = \mu B \circ (\eta B \circ Ff) \\
\Rightarrow\ &\quad \big[\text{ the two middle terms are equal} \\
&Hf \circ (\mu A \circ \eta A) = (\mu B \circ \eta B) \circ Ff \\
\Rightarrow\ &\quad \big[\text{ Definition 5.3.4} \\
&Hf \circ \mu\eta A = \mu\eta B \circ Ff \\
\Rightarrow\ &\mu\eta : F \dot{\longrightarrow} H
\end{aligned}$$

□

Chapter 6

Monads

6.1 Monads in theory

6.1.1 Definition Let $\mathcal{C}$ be a category, and $F : \mathcal{C} \to \mathcal{C}$. Consider two natural transformations $\eta : I_{\mathcal{C}} \dot{\to} F$ and $\mu : F^2 \dot{\to} F$.

The triple (F, η, μ) is called a **monad**, iff

$$\mu(F\mu) = \mu(\mu F) \quad \text{and} \quad \mu(F\eta) = \mathrm{id}_F = \mu(\eta F) \ .$$

(Mind, that the parenthesis group composition of natural transformations and functors. They do not refer to function application. This is obvious due to the types of the expressions in question.)

The transformations η and μ are somewhat contrary: While η adds one level of structure (i.e., functor application), μ removes one. Note, however, that η transforms to F, while μ transforms from F^2. This is due to the fact that claiming the existence of a transformation from a functor F to the identity $I_{\mathcal{C}}$ would be too restrictive:

Consider the set category $\mathcal{S}$ and the list endofunctor L. Any transformation ϵ from L to $I_{\mathcal{S}}$ has to map every object A to a morphism ϵ_A, which in turn maps the empty list to some element in A, i.e.,

$$\begin{array}{ll} & \epsilon : L \to I_{\mathcal{S}} \\ \Rightarrow & \forall\ A \in \mathcal{O}_{\mathcal{S}} \quad \epsilon_A : LA \to A \\ \Rightarrow & \forall\ A \in \mathcal{O}_{\mathcal{S}} \quad \exists\ c \in A \quad \epsilon_A[\,] = c \ , \end{array}$$

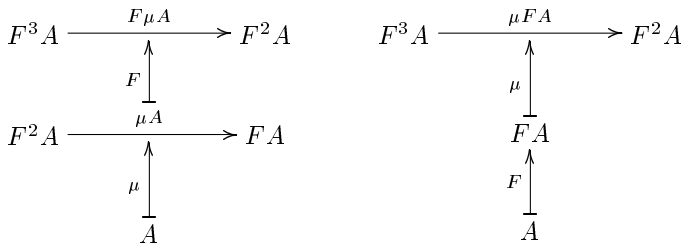
where $[\,]$ denotes the empty list. This, however, implies

$$\forall\ A \in \mathcal{O}_{\mathcal{S}} \quad A \neq \emptyset \ .$$

The following drawings are intended to clarify Definition 6.1.1: The first equation of the definition is equivalent to

$$\forall\ A \in \mathcal{O} \quad \mu A \circ F\mu A = \mu A \circ \mu F A \ .$$

So what are $F\mu A$ and $\mu F A$? You can find them at the top of these drawings:



Since application of μA unnests a nested structure by one level, $F\mu A$ pushes application of this unnesting one level into the nested structure. We need at least two levels of nesting to apply μA.

So we need at least three levels of nesting to push the application of μA one level in. This justifies the type of $F\mu A$.

The morphism μFA, however, applies the reduction on the outer level. The FA only assures that there is one more level on the inside which, however, is not touched.

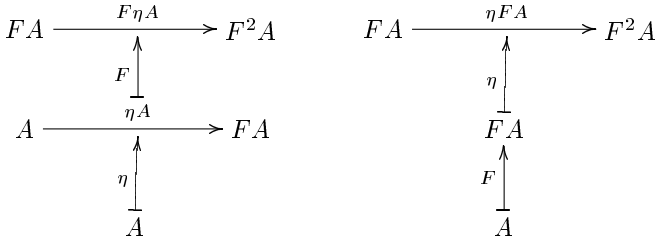
Hence, $F\mu A$ and μFA depict the two possibilities to flatten a three-level nested structure into a two-level nested structure through application of a mapping that flattens a two-level nested structure into an one-level nested structure.

The statement $\mu A \circ F\mu A = \mu A \circ \mu FA$ says, that after another step of unnesting (i.e., μA), it is irrelevant which of the two inner structure levels has been removed by prior unnesting (i.e., $F\mu A$ or μFA).

Let us have a look at the second equation as well. It is equivalent to

$$\forall\, A \in \mathcal{O}_{\mathcal{C}} \quad \mu A \circ F\eta A = \mathrm{id}_{FA} = \mu A \circ \eta FA \ .$$

Again, we examine $F\eta A$ and ηFA, which are at the top of the drawings.



Since ηA adds one level of nesting to A, the morphism $F\eta A$ imposes this lifting inside a flat structure (FA), yielding a nested structure F^2A.

Also, ηFA adds one level of nesting, however on the outside of a flat structure.

I.e., similar to the situation above, $F\eta A$ and ηFA reflect the two ways to add one level of nesting to a flat structure: It can be done by lifting the whole structure, or by lifting the things within the structure.

So $\mu A \circ F\eta A = \mathrm{id}_{FA} = \mu A \circ \eta FA$ states, just as for the first equation, that after unnesting (i.e., μA) it is not relevant whether additional structure has been added on the outside (ηFA) or the inside ($F\eta A$) of a flat structure FA. Additionally, it says that nesting followed by unnesting, is the identity.

6.2 Monads in Haskell

The list structure (we use L to denote the according functor) provided by the Haskell language is probably the most intuitive example to explain a monad. That it is a monad indeed is stated in Lemma 6.2.1.

Imagine a list of lists of lists of natural numbers (i.e., $L^3\mathbb{N}$). With a transformation like list concatenation ($\mu : L^2 \to L$) we can choose between two alternatives of flattening, considering L^3 to be a "list of (list of lists)" or a "(list of lists) of lists":

μLA applies the *concatenation of a list of lists* to the outermost *list of lists*. Here, A refers to the object $L\mathbb{N}$ corresponding to the third-level list structure, which is the topmost level in the nested structure that is not effected by the transformation.

Application of μLA returns a new list, containing all the unchanged innermost lists. You can try (✎)

```
concat [  [ [1,2], [3,4] ]
       ,  [ [5,6], [7,8] ]
       ]
```

in your Haskell interpreter.

$L\mu A$ lifts the *concatenation of a list of lists* into the outermost list, applying it to each of the second-level *lists of lists*. Here, A refers to the object $\mathbb{N}$, corresponding to the members of the third

level list structure. $\mathbb{N}$ is the topmost level in the nested structure that is not effected by the transformation.

Application of $L\mu A$ returns the outermost list with its elements replaced by the results of applying μA on them. You can observe this using (✎) the Haskell code

```
fmap concat [  [ [1,2], [3,4] ]
            ,  [ [5,6], [7,8] ]
            ]
```

with your interpreter.

The monad property $\mu(F\mu) = \mu(\mu F)$ can be observed by applying `concat` to the results of the two expressions given above. Both yield the same result.

Now, consider the transformation $\eta : I_{\mathcal{C}} \to L$, which, parametrised with an object A, maps a member from A to the singleton list in LA containing that member. Again, we face two alternatives:

ηLA applies the *list making* to the whole input list, returning a list which simply contains the original input list as sole member. Feed (✎) the code

```
(\x -> [x]) [1,2,3]
```

to Haskell.

$L\eta A$ applies the *list making* to each element in the input list, returning the input list with its members replaced by the according singleton lists. You can type (✎)

```
fmap (\x -> [x]) [1,2,3]
```

to verify this.

The monad property $\mu(L\eta) = \mathrm{id}_L = \mu(\eta L)$ can be observed (✎) by applying `concat` to the results of the two expressions given above. Both yield the same result.

6.2.1 Lemma Haskell's List structure L, together with concatenation μ and the creation of singleton lists η, forms a monad (L, μ, η).

6.2.2 Proof We already know that L is a functor (Lemma 4.3.1). The remaining work to do is: Prove the naturality of μ and η, and show that the monad property holds.

We use the Haskell notation for lists, i.e., $[\,]$ denotes the empty list, and $x : x'$ denotes the list x' prepended with the element x. The definition of `fmap` yields

$$(Lf)[\,] = [\,]$$
$$(Lf)(x : x') = x : (Lf)x' \ .$$

Note, that `concat` is defined in terms of `foldr` and a binary concatenation operator ($+\!\!+$) we consider primitive:

```
concat = foldr (++) [] .
```

From this, we conclude $\mu_B(x : x') = x +\!\!+ \mu_B x'$ and $\mu_B[\,] = [\,]$.

▷ **Part I** To prove that μ is a natural transformation, we have to show that

$$\forall\, f : A \to B \quad \mu B \circ L^2 f = Lf \circ \mu A$$

holds. We use induction on the structure of the list x.
Let $x = [\,] \in L^2 A$. Then, we observe that

$$(\mu B \circ L^2 f)[\,] = (Lf \circ \mu A)[\,]$$

holds by looking at the definitions of `fmap` (L) and `concat` (μ_A and μ_B).

Let $x = y : y'$, where $y \in LA$ and $y' \in L^2A$.

$$
\begin{array}{ll}
 & (\mu_B \circ L^2 f)x \\
= & \\
 & \mu_B((L^2 f)(y : y')) \\
= & \quad [\text{ definition of } \texttt{fmap} \\
 & \mu_B((Lf)y : (L^2 f)y') \\
= & \quad [\text{ definition of } \texttt{concat} \\
 & (Lf)y +\!\!+ \mu_B((L^2 f)y') \\
= & \quad [\text{ induction: } (\mu_B \circ L^2 f)y' = (Lf \circ \mu_A)y' \\
 & (Lf)y +\!\!+ (Lf)(\mu_A y') \\
= & \\
 & (Lf)(y +\!\!+ \mu_A y') \\
= & \quad [\text{ definition } \texttt{concat} \\
 & (Lf)(\mu_A(y : y')) \\
= & \\
 & (Lf \circ \mu_A)x
\end{array}
$$

▷ **Part II** To prove that η is a natural transformation, we have to show that

$$\forall\, f : A \to B \quad \eta B \circ f = Lf \circ \eta A$$

holds. The proof reads

$$
\begin{array}{ll}
 & (\eta_B \circ f)x \\
= & \\
 & \eta_B(fx) \\
= & \\
 & [fx] \\
= & \\
 & (Lf)[x] \\
= & \\
 & (Lf)(\eta_A x) \\
= & \\
 & (Lf \circ \eta_A)x \ .
\end{array}
$$

▷ **Part III** Now we show that the monad properties for lists hold.
Again, we use induction on the structure of a list x.

▷ **Part III.a** Proof that $\mu(L\mu) = \mu(\mu L)$.
Let $x = [\,] \in F^3 A$.

$$
\begin{array}{ll}
 & (\mu_A \circ L\mu_A)[\,] \\
= & \\
 & \mu_A((L\mu_A)[\,]) \\
= & \\
 & \mu_A[\,] \\
= & \\
 & \mu_A(\mu_{LA}[\,]) \\
= & \\
 & (\mu_A \circ \mu_{LA})[\,] \ .
\end{array}
$$

Let $x = y : y'$, where $y \in L^2 A$ and $y' \in L^3 A$.

$$
\begin{array}{ll}
 & (\mu_A \circ L\mu_A)x \\
= & \\
 & \mu_A((L\mu_A)(y : y')) \\
= & \quad [\text{ definition of } \texttt{fmap} \\
 & \mu_A(\mu_A y : (L\mu_A)y') \\
= & \quad [\text{ definition of } \texttt{concat} \\
 & \mu_A y +\!\!+ \mu_A((L\mu_A)y') \\
= & \quad [\text{ induction} \\
 & \mu_A y +\!\!+ \mu_A(\mu_{LA} y') \\
= & \quad [\text{ definition of } \texttt{concat}
\end{array}
$$

$$
\begin{array}{ll}
= & \mu_A(y +\!\!+ \mu_{LA} y') \\
= & \mu_A(\mu_{LA}(y : y')) \\
& (\mu_A \circ \mu_{LA})x \ .
\end{array}
$$

▷ **Part III.b** Proof that $\mu(L\eta) = \mathrm{id}_L = \mu(\eta L)$.
Let $x = [\,] \in LA$.

$$
\begin{array}{ll}
= & (\mu_A \circ L\eta_A)[\,] \\
& \mu_A((L\eta_A)[\,]) \\
= & \qquad [\text{ definition of } \texttt{fmap} \\
& \mu_A[\,] \\
= & \qquad [\text{ definition of } \texttt{concat} \\
& [\,] \\
= & \qquad [\text{ definition of } \texttt{concat} \\
& \mu_A[[\,]] \\
= & \mu_A(\eta_{LA}[\,]) \\
= & (\mu_A \circ \eta_{LA})[\,]
\end{array}
$$

Let $x = y : y'$, where $y \in A$ and $y' \in LA$.

$$
\begin{array}{ll}
= & (\mu_A \circ L\eta_A)x \\
& \mu_A((L\eta_A)(y : y')) \\
= & \qquad [\text{ definition of } \texttt{fmap} \\
& \mu_A(\eta_A y : (L\eta_A)y') \\
= & \qquad [\text{ definition of } \texttt{concat} \\
& \eta_A y +\!\!+ \mu_A((L\eta_A)y') \\
= & \qquad [\text{ induction: } \mu_A \circ L\eta_A = \mathrm{id}_{LA} \\
& \eta_A y +\!\!+ y' \\
= & y : y' \\
= & \mu_A[y : y'] \\
= & \mu_A(\eta_{LA}(y : y')) \\
= & (\mu_A \circ \eta_{LA})x \ .
\end{array}
$$

□

6.3 An alternative definition of the monad

While the previous definition of the monad structure is quite intuitive, there is another one and, in fact, that is the one Haskell uses for its `Monad` class.

First, we use the monad as introduced in Definition 6.1.1, to define a *bind* operator, and we prove *the three monad laws* to hold for this definition. Then, we show that the three monad laws alone imply all the properties required to form a monad, hence yield an alternative definition.

6.3.1 Definition Given (F, η, μ), we define the binary infix operator **bind** by

$$
\begin{array}{lccccc}
\gg\!= & : & FA & \times & (A \to FB) & \longrightarrow & FB \\
& & x & , & f & \longmapsto & (\mu_B \circ Ff)x \ .
\end{array}
$$

Note, that we assume x to be a member of FA, which restricts us to categories which support this — like, e.g., the category of sets, or $\mathcal{H}$. In literature not related to the Haskell language a point-free variant of the bind operator, called *Kleisli star*, is used:

6.3.2 Definition The unary postfix operator **Kleisli star**, a point-free version of the bind operator, is defined by

$$\begin{array}{rccc} {}^* & : & (A \to FB) & \longrightarrow & (FA \to FB) \\ & & f & \longmapsto & \mu_B \circ Ff \ . \end{array}$$

Obviously $\forall\, x \in FA\,;\ f : A \to FB \quad x \gg\!= f = f^* x$.

In fact, the bind operator is a special case of the Kleisli star, restricted to categories where the objects have members. Ignoring the lack of elegance and generality, we stick to the bind notation in the following, since this matches the Haskell notation. However, it is easy (✎) to reformulate the following statements and proofs to the more general Kleisli notation.

The second argument of $\gg\!=$ is a function f, which adds some structure F to what it returns. Intuitively, as defined above, the bind operator lifts the passed function f, applies it to the object $x \in FA$, and then applies μ to remove one level of structure nesting.

The bind operator is used to formulate *the three monad laws*. We give them in the form of a lemma and prove their correctness using the monad properties given in Definition 6.1.1.

6.3.3 Lemma The three monad laws are:

1. η_A resembles a left identity with respect to the bind operator:
   $$\forall\, f : A \to FB\,;\ x \in A \quad \eta_A x \gg\!= f = fx \ .$$
2. η_A is a right identity with respect to the bind operator:
   $$\forall\, x \in FA \quad x \gg\!= \eta_A = x$$
3. The bind operator is quite close to being associative:
   $$\forall\, x \in FA\,;\ f : A \to FB\,;\ g : B \to FC$$
   $$(x \gg\!= f) \gg\!= g = x \gg\!= (y \mapsto fy \gg\!= g)$$

Just as a hint for the exercise: Using Kleisli notation, the three monad laws read

1. $\forall\, f : A \to FB \quad f^* \circ \eta_A = f$
2. $\forall\, A \in \mathcal{O} \quad \eta_A^* = \mathrm{id}_{FA}$
3. $\forall\, f : A \to FB\,;\ g : B \to FC \quad g^* \circ f^* = (g^* \circ f)^*$

6.3.4 Proof of Lemma 6.3.3.

▷ **Part I** Let $f : A \to FB$, and $x \in A$.

$$\begin{array}{ll} & \eta_A x \gg\!= f \\ = & \quad [\text{ bind operator, Definition 6.3.1} \\ & (\mu_B \circ Ff)(\eta_A x) \\ = & (\mu_B \circ Ff \circ \eta_A)x \\ = & \quad [\text{ naturality of } \eta \text{ means } \eta_{FB} \circ If = Ff \circ \eta_A \\ & (\mu_B \circ \eta_{FB} \circ If)x \\ = & \quad [\text{ using the monad property } \mu(\eta F) = \mathrm{id}_F \\ & (\mathrm{id}_{FB} \circ f)x \\ = & fx \end{array}$$

▷ **Part II** Let $x \in FA$.

$$\begin{array}{ll} & x \gg\!= \eta_A \\ = & \quad [\text{ using Definition 6.3.1 of the bind operator} \\ & (\mu_A \circ F\eta_A)x \\ = & \quad [\text{ monad property } \mu(F\eta) = \mathrm{id}_F \end{array}$$

$$
\begin{array}{ll}
= & \mathrm{id}_{FA}\, x \\
& x
\end{array}
$$

▷ **Part III** Let $x \in FA$, $f : A \to FB$, and $g : B \to FC$.

$$
\begin{array}{ll}
& (x \gg\!= f) \gg\!= g \\
= & \qquad [\text{ using Definition 6.3.1 of the bind operator} \\
& ((\mu_B \circ Ff)x) \gg\!= g \\
= & \qquad [\text{ dito} \\
& (\mu_C \circ Fg)((\mu_B \circ Ff)x) \\
= & (\mu_C \circ Fg \circ \mu_B \circ Ff)x \\
= & \qquad [\text{ naturality of } \mu \text{ means } \mu_{FC} \circ F^2 g = Fg \circ \mu_B \\
& (\mu_C \circ \mu_{FC} \circ F^2 g \circ Ff)x \\
= & \qquad [\text{ using the monad property } \mu(\mu F) = \mu(F\mu) \\
& (\mu_C \circ F\mu_C \circ F^2 g \circ Ff)x \\
= & \qquad [\ F \text{ is a functor, so it distributes under composition.} \\
& (\mu_C \circ F(\mu_C \circ Fg \circ f))x \\
= & \qquad [\text{ the bind operator again} \\
& x \gg\!= \mu_C \circ Fg \circ f \\
= & \qquad [\text{ build a } \lambda\text{-expression} \\
& x \gg\!= (y \mapsto (\mu_C \circ Fg \circ f)y) \\
= & x \gg\!= (y \mapsto (\mu_C \circ Fg)(fy)) \\
= & \qquad [\text{ the bind operator again} \\
& x \gg\!= (y \mapsto fy \gg\!= g)
\end{array}
$$

□

Note, that the three monad laws do not make use of μ or $F_{\mathcal{M}}$. In the following, it turns out that defining (F, η, μ) is equivalent to defining $(F_{\mathcal{O}}, \eta, \gg\!=)$, i.e., each of these tuples can be defined in terms of the other: The one direction is obvious using Definition 6.3.1, the reverse is given in Definition 6.3.6. Moreover, Theorem 6.3.5 states that the three monad laws are another way to formulate the previously given monad properties (Definition 6.1.1).

6.3.5 Theorem Let $\mathcal{C}$ be a category, and $F_{\mathcal{O}} : \mathcal{O} \to \mathcal{O}$ an arbitrary object mapping. Consider a function (like a transformation)

$$
\begin{array}{lccc}
\eta & : & \mathcal{O} & \longrightarrow & \mathcal{M} \\
 & & A & \longmapsto & \eta_A
\end{array}
$$

with $\eta_A : A \xrightarrow[\mathcal{C}]{} FA$, and an operator

$$
\gg\!= : F_{\mathcal{O}} A \times (A \to F_{\mathcal{O}} B) \longrightarrow F_{\mathcal{O}} B \ .
$$

Then, the triple $(F_{\mathcal{O}}, \eta, \gg\!=)$ forms a monad, iff the three monad laws (as given in Lemma 6.3.3) hold.

Note, that η is not required to be a natural transformation. This turns out to be a result of the three monad laws.

6.3.6 Definition In the situation of Theorem 6.3.5, we can define μ and $F_{\mathcal{M}}$ for all $A \in \mathcal{O}_{\mathcal{C}}$ and all $f : A \xrightarrow[\mathcal{C}]{} B$ by

$$
\begin{array}{lccc}
\mu_A & : & F_{\mathcal{O}}^2 A & \longrightarrow & F_{\mathcal{O}} A \\
 & & x & \longmapsto & x \gg\!= \mathrm{id}_{F_{\mathcal{O}} A}
\end{array}
$$

and

$$\begin{array}{rccc} F_{\mathcal{M}} f & : & F_{\mathcal{O}} A & \longrightarrow & F_{\mathcal{O}} B \\ & & x & \longmapsto & x \gg\!= \eta_B \circ f \ . \end{array}$$

Again, we write F for $F_{\mathcal{O}}$ as well as for $F_{\mathcal{M}}$, since they hardly can be confused. Definition 6.3.6 is used in the following, so bear in mind that we have to prove the monad properties of the (F, η, μ) tuple just defined, by using the three monad laws we claimed to hold in Theorem 6.3.5.

6.3.7 Proof of Theorem 6.3.5. Due to Lemma 6.3.3, we only need to show the functor properties of F, that η is a natural transformation, and that we can conclude the monad properties given in Definition 6.1.1 from the three monad laws given in Lemma 6.3.3.

▷ **Part I** First, we show the functor properties of F.

▷ **Part I.a** That $F_{\mathcal{M}}$ preserves type information, i.e., that

$$\forall\, f : A \to B \quad Ff : FA \to FB$$

holds, follows directly from its definition.

▷ **Part I.b** $F_{\mathcal{M}}$ preserves identities, since for all $x \in FA$

$$(F\,\mathrm{id}_A)x = x \gg\!= \eta_A \circ \mathrm{id}_A = x \gg\!= \eta_A = x = \mathrm{id}_{FA}\, x$$

holds.

▷ **Part I.c** For the distribution of $F_{\mathcal{M}}$ under composition we fix arbitrary $x \in FA$, $f : A \to B$ and $g : B \to C$. Then

$$\begin{array}{ll} & (Fg \circ Ff)x \\ = & \quad [\text{ Definition 6.3.6 of } F_{\mathcal{M}} \text{ applied twice} \\ & (x \gg\!= \eta_B \circ f) \gg\!= \eta_C \circ g \\ = & \quad [\text{ the 3rd monad law} \\ & x \gg\!= (y \mapsto (\eta_B \circ f)y \gg\!= \eta_C \circ g) \\ = & \quad [\text{ the 1st monad law} \\ & x \gg\!= (y \mapsto (\eta_C \circ g \circ f)y) \\ = & \quad [\text{ removing the } \lambda\text{-expression} \\ & x \gg\!= \eta_C \circ g \circ f \\ = & \quad [\text{ Definition 6.3.6 of } F_{\mathcal{M}} \text{ applied to } (g \circ f) \\ & (F(g \circ f))x \ . \end{array}$$

Thus, $\forall\, f : A \to B\,;\ g : B \to C \quad F(g \circ f) = Ff \circ Gf$.

Together, we conclude that F is an endofunctor in $\mathcal{C}$.

▷ **Part II** Since F is a functor, η is a transformation from $I_{\mathcal{C}}$ to F by definition. To show its naturality, choose arbitrary $x \in A$ and $f : A \to B$. Then

$$\begin{array}{ll} & (Ff \circ \eta_A)x \\ = & Ff(\eta_A x) \\ = & \quad [\text{ Definition 6.3.6 of } F_{\mathcal{M}} \\ & \eta_A x \gg\!= \eta_B \circ f \\ = & \quad [\text{ the 1st monad law} \\ & (\eta_B \circ f)x \\ = & \quad [\text{ inserting the identity functor} \\ & (\eta_B \circ If)x \end{array}$$

Thus, $\eta : I_{\mathcal{C}} \dot{\longrightarrow} F$.

▷ **Part III** To show $\mu(\eta F) = \mathrm{id}_F = \mu(F\eta)$, we prove the two equations separately for fixed arbitrary $A \in \mathcal{O}$, and $x \in FA$.

▷ **Case III.a** For the left hand side equation the proof reads

$$
\begin{aligned}
& (\mu A \circ \eta FA)x \\
=\ & \mu A((\eta FA)x) \\
=\ & \quad \big[\text{Definition 6.3.6 of } \mu \\
& ((\eta FA)x) \gg\!= \mathrm{id}_{FA} \\
=\ & \quad \begin{bmatrix} \text{Use that } (\eta A')x \gg\!= f = fx \text{ for } x \in A',\ f : A' \to FB', \text{ and assign } A' := FA, \\ B' := A \text{ and } f := \mathrm{id}_{FA}. \end{bmatrix} \\
=\ & \mathrm{id}_{FA}\, x \\
=\ & x \ .
\end{aligned}
$$

▷ **Case III.b** For the right hand side equation we calculate

$$
\begin{aligned}
& (\mu A \circ F\eta A)x \\
=\ & (\mu A)((F\eta A)x) \\
=\ & \quad \big[\text{Definition 6.3.6 of } F_{\mathcal{M}} \text{ with } f := \eta_A \text{ reads } (F\eta_A)y = y \gg\!= \eta_{FA} \circ \eta_A, \text{ for } y \in FA. \\
& (\mu A)(x \gg\!= \eta FA \circ \eta A) \\
=\ & \quad \big[\text{Definition 6.3.6 of } \mu \\
& (x \gg\!= \eta FA \circ \eta A) \gg\!= \mathrm{id}_{FA} \\
=\ & \quad \big[\text{the 3rd monad law} \\
& x \gg\!= (y \mapsto (\eta FA \circ \eta A)y \gg\!= \mathrm{id}_{FA}) \\
=\ & \quad \big[\text{Definition 6.3.6 of } \mu \text{ used backwards} \\
& x \gg\!= (y \mapsto \mu A((\eta FA \circ \eta A)y)) \\
=\ & \quad \big[\text{remove the } \lambda\text{-expression} \\
& x \gg\!= \mu A \circ \eta FA \circ \eta A \\
=\ & \quad \big[\text{Part III.a: } \mu(\eta F) = \mathrm{id}_F \\
& x \gg\!= \eta A \\
=\ & \quad \big[\text{the 2nd monad law} \\
& x
\end{aligned}
$$

▷ **Part IV** To show $\mu(\mu F) = \mu(F\mu)$ we fix arbitrary $A \in \mathcal{O}$ and $x \in F^3A$. Then,

$$
\begin{aligned}
& (\mu A \circ \mu FA)x \\
=\ & (\mu A)((\mu_{FA})x) \\
=\ & \quad \big[\text{Definition 6.3.6 of } \mu, \text{ using } FA \text{ instead of } A \text{ there} \\
& (\mu A)(x \gg\!= \mathrm{id}_{F^2A}) \\
=\ & \quad \big[\text{definition of } \mu \text{ again} \\
& (x \gg\!= \mathrm{id}_{F^2A}) \gg\!= \mathrm{id}_{FA} \\
=\ & \quad \big[\text{the 3rd monad law} \\
& x \gg\!= (y \mapsto \mathrm{id}_{F^2A}\, y \gg\!= \mathrm{id}_{FA}) \\
=\ & x \gg\!= (y \mapsto y \gg\!= \mathrm{id}_{FA}) \\
=\ & \quad \big[\text{definition of } \mu \\
& x \gg\!= (y \mapsto (\mu A)y) \\
=\ & \quad \big[\text{removing the } \lambda\text{-expression} \\
& x \gg\!= \mu A \\
=\ &
\end{aligned}
$$

$$
\begin{aligned}
& x \gg\!= \mathrm{id}_{FA} \circ \mu A \\
= & \quad [\text{ Part III.a: } \mu(\eta F) = \mathrm{id}_F \\
& x \gg\!= \mu A \circ \eta F A \circ \mu A \\
= & \quad [\text{ building a } \lambda\text{-expression} \\
& x \gg\!= (y \mapsto (\mu A)((\eta F A \circ \mu A) y)) \\
= & \quad [\text{ definition of } \mu \\
& x \gg\!= (y \mapsto (\eta F A \circ \mu A) y \gg\!= \mathrm{id}_{FA}) \\
= & \quad [\text{ the 3rd monad law} \\
& (x \gg\!= \eta F A \circ \mu A) \gg\!= \mathrm{id}_{FA} \\
= & \quad [\text{ definition of } \mu \\
& (\mu A)(x \gg\!= \eta F A \circ \mu A) \\
= & \quad [\text{ definition of } F_{\mathcal{M}}, \text{ used backwards, with } f := \mu A \\
& (\mu A)((F \mu A) x) \\
= & \\
& (\mu A \circ F \mu A) x
\end{aligned}
$$

□

6.4 Haskell's monad class and do-notation

Haskell provides the programmer with a class `Monad`, which uses the alternative definition given above.

```
class Monad m :: (* -> *) where
  (>>=) :: forall a b. m a -> (a -> m b) -> m b
  return :: forall a. a -> m a;
```

In fact, there are two more functions defined —namely `(>>)` and `fail`— but we do not discuss them here. The `return` function is Haskell's equivalent to the natural transformation η. The operator `>>=` simply is the bind operator. The type variable `m` is bound to a unary type constructor: the functor.

As for the `Functor` class, being a member of `Monad` alone does not assure adherence to the monad laws. This has to be checked by the programmer in advance. Luckily, Theorem 6.3.5 saves us from doing this again, if we did it before.

Definition 6.3.1 directly shows how we could make Haskell's List structure a member of the `Monad` class — if it was not already:

```
instance Monad List where
  return x = [x]
  x >>= f = (concat . fmap f) x
```

Also, we could make `Maybe` a member of the `Monad` class:

```
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

It is a nice —and easy— exercise, to verify the three monad laws for the Maybe monad as given above.

Membership of the `Monad` class allows usage of the *do-notation*. This is a syntactic construct provided by the Haskell compiler which resembles imperative programming.

6.4.1 Definition The **do-notation** is defined (Section 3.14 of [5]) using the following recursive set of rules (for brevity, we skip the definition of *let* clauses). To form a do-expression, the `do`

keyword operates on a sequence of terms $t_0; \ldots; t_n$, where t_n is a Haskell expression, and the t_i with $0 \leq i < n$ are either Haskell expressions as well, or pattern matching expressions $v_i \leftarrow e_i$, where v_i are patterns introducing new variable names.

1. If the list of terms contains only one term —which then must be an expression— the do is simply removed:
   $$\mathrm{do}\ \{\ e\ \} \quad := \quad e\ .$$
2. If the first term is a pattern matching expression $v \leftarrow e$, a λ-expression is created binding the pattern v in a do-expression containing the remaining terms. The result of the expression e is bound to the new λ-expression:
   $$\mathrm{do}\ \{\ v \leftarrow e;\ t_1;\ \ldots;\ t_n\ \} \quad := \quad e \gg\!= (v \mapsto \mathrm{do}\ \{\ t_1;\ \ldots;\ t_n\ \})\ .$$
   Note, how this resembles *variable assignment* in an imperative programming language, and also justifies the name of the bind operator.
3. If there are multiple terms in the sequence, but the first one does not contain a pattern matching, we use
   $$\mathrm{do}\ \{\ e;\ t_1;\ \ldots;\ t_n\ \} \quad := \quad e \gg\!= (\_ \mapsto \mathrm{do}\ \{\ t_1;\ \ldots;\ t_n\ \})\ ,$$
   which means that the value of e is not used in the remaining term.

Above, the sequence of terms is surrounded by curly braces, but it is also possible to use Haskell's two-dimensional syntax.

6.5 First step towards IO

As introduction, we build a monad that allows us to model IO operations. Assume our running program to be connected to an input and an output character stream. We encapsulate their state inside a monad structure and use imperative programming style to manipulate them. Hence, in the following example, we do not use the "real" IO monad.

The state of the input and output streams is maintained in a pair of strings

```
(String,String)
```

where the former string models the characters waiting in the input stream, and the latter represents what was written to the output stream. Hence, a value of

```
("foo","bar")
```

means, that the output `bar` has been written already, and that `foo` has not been read from the input yet.

Obviously, we can model IO by passing around a parameter containing the *state of the world* outside our Haskell program. In fact, this is what we'll do, however encapsulating the state in a monad structure.

Functions that change the *state of the world*, i.e., read or write the streams, are called *state transformations*. We use (✎) a data type

```
data ST a = ST( (String,String) -> ((String,String),a) )
```

to model them. The state transformations are equipped with a type parameter `a`. This is required, because we want the state transformations to return data, depending on the change performed to the world. E.g., reading a character from the input not only changes the input stream (by removing that character), but also *returns* that character. Hence, we refer to `a` as **return value** of a transformation.

With this, we can already define (✎) our primitive input and output routines:

```
getc :: ST Char
getc = ST( λ((i:is),os) -> ((is,os),i) )

putc :: Char -> ST ()
putc c = ST( λ(is,os) -> ((is,os++[c]),()) )
```

The `getc` function takes the first character from the input stream, and returns it in `i` — note the type parameter `Char` passed to `ST`. Clearly, `getc` transforms the input state, by removing one character from the input stream. The output stream remains unchanged.

Writing to the output stream appends the character passed to the `putc` function to the already written output `os`. We don't want to return anything, which we model by returning Haskell's unit type value `()`.

So how do we define the monad? Since we want to use do-notation, we make the state transformation an instance of the monad class

```
instance Monad ST where
...
```

The natural transformation η is used to *return* a value, i.e., ηA maps a value $x \in A$ to a state transformation which returns x without altering the *state of the world* s_0:

$$\eta_A x := s_0 \mapsto (s_0, x) \ .$$

In Haskell syntax (✎), this reads

```
return :: a -> ST a
return val = ST (λs0 -> (s0,val)) .
```

The bind operator is a bit more complex: Assuming S to be the functor that embodies the structure `ST` of a state transformation, for a morphism $f : A \to SB$ and a transformation $t_0 \in SA$ (which returns a value in A) we define

$$t_0 \gg\!= f := s_0 \mapsto t_1 s_1 \quad \text{where} \quad (s_1, v) = t_0 s_0 \text{ and } t_1 = fv \ .$$

To understand this, observe that the binding of a state transformation t_0 to a function $f : A \to SB$ returns a new state transformation, which is assembled as follows:

The state transformation t_0 is applied to the input state s_0 of the generated transformation, returning a new state s_1 and a return value v. Applying f to this value leads to a new state transformation t_1 which is then applied to the new state s_1.

It becomes clear now, how the return value is used: Transforming the input state s_0 using t_0 returns a value in v which is passed to f to create the new state transformation t_1.

In Haskell syntax, the bind operator reads as follows:

```
(>>=) ::  ST a -> (a -> ST b) -> ST b
(ST t0) >>= f = ST( λs0 -> let (s1,val) = t0 s0
                               (ST t1) = f val
                               in t1 s1
                  )
```

6.5.1 Theorem The triple `(ST,return,>>=)` is a monad.

6.5.2 Notation We introduce a bit of notation to increase readability of the following proof: Let $[q]_k$ address the k-th member of a tuple q if it contains at least k entries, i.e.,

$$\forall\, k, n \in \mathbb{N};\ 1 \le k \le n \quad [(a_1, \ldots a_n)]_k = a_k \ .$$

We use this notation to access the new state and the return value of a state transformation. The definition of the $\gg\!=$ operator for the `ST` structure now reads

$$= \begin{array}{l} t \gg\!= f \\ s \mapsto (\ f[ts]_2\ )\ [ts]_1 \end{array}$$

where $[ts]_1$ refers to the new state, and $[ts]_2$ refers to the return value.

6.5.3 Proof that S forms a monad.

▷ **Part I** The first two parts are quite trivial. But let us get used to the $[\cdot]_k$-notation. We see the first monad law by

$$
\begin{array}{ll}
 & t \gg\!= \eta_A \\
= & \\
 & s \mapsto (\eta_A [ts]_2)[ts]_1 \\
= & \quad [\text{ definition of } \eta \\
 & s \mapsto (s' \mapsto (s', [ts]_2))[ts]_1 \\
= & \\
 & s \mapsto ([ts]_1, [ts]_2) \\
= & \\
 & s \mapsto ts \\
= & \\
 & t \ .
\end{array}
$$

▷ **Part II** And the second law is shown by

$$
\begin{array}{ll}
 & \eta_A x \gg\!= f \\
= & \quad [\text{ definition of } \gg\!= \text{ using Notation 6.5.2} \\
 & s \mapsto (f[(\eta_A x)s]_2)[(\eta_A x)s]_1 \\
= & \quad [\text{ definition of } \eta \\
 & s \mapsto (f[(s' \mapsto (s', x))s]_2)[(s' \mapsto (s', x))s]_1 \\
= & \\
 & s \mapsto (f[(s, x)]_2)[(s, x)]_1 \\
= & \\
 & s \mapsto (fx)s \\
= & \\
 & fx \ .
\end{array}
$$

▷ **Part III** The least intuitive part is probably the third monad law:

$$
\begin{array}{ll}
 & (t \gg\!= f) \gg\!= g \quad = \quad t \gg\!= (y \mapsto fy \gg\!= g) \\
\Leftarrow & \quad [\text{ definition of } \gg\!= \text{ applied to its right occurrence on the left hand side} \\
 & s \mapsto (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad t \gg\!= (y \mapsto fy \gg\!= g) \\
\Leftarrow & \quad [\text{ definition of } \gg\!= \text{ applied to its left occurrence on the right hand side} \\
 & s \mapsto (g[(t \gg\!= f)s]_2)[t \gg\!= f)s]_1 \quad = \quad s \mapsto (f[ts]_2 \gg\!= g)[ts]_1 \\
\Leftarrow & \quad [\text{ generalising over all } s \\
 & (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad (f[ts]_2 \gg\!= g)[ts]_1 \\
\Leftarrow & \quad [\text{ definition of } \gg\!= \text{ applied on the right hand side} \\
 & (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad (s_2 \mapsto (g[(f[ts]_2)s_2]_2)[(f[ts]_2)s_2]_1)[ts]_1 \\
\Leftarrow & \quad [\text{ resolving the } \lambda\text{-expression on the right hand side} \\
 & (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad (g[(f[ts]_2)[ts]_1]_2)[(f[ts]_2)[ts]_1]_1 \\
\Leftarrow & \quad [\text{ observing the } (g[X]_2)[Y]_1 \text{ skeleton on both sides} \\
 & (t \gg\!= f)s \quad = \quad (f[ts]_2)[ts]_1 \ .
\end{array}
$$

This is the definition of the $\gg\!=$ operator after applying the λ-expression.

□

Now let us have a look at some programs written in the imperative programming style offered by the do-notation. To discuss the code, we use imperative style language, like “program” or “assign”.

✎ **First example** The code fragment

```
t0 = getVal (do initialise
            )
```

is the shortest program simulating IO we can write so far. It does nothing, however we have to understand this skeleton, which is also used below. The auxiliary functions `getVal` and `initialise` are required in our model:

Since we only model IO operations —i.e., our program does not do real IO yet— we somehow need to get an initial state of the input and output streams. This is done by

```
initialise :: ST ()
initialise = ST(λ_ -> (("foo",""),()))
```

which sets the input stream to `"foo"`, and the output stream to be empty. This "setting the state" is performed by creating a transformation, which ignores its input (hence `λ_ -> ...`) and always returns the same state. Since initialisation shall not return a value, we return `()`.

When our program in do-notation has been executed, we access the state of the program via

```
getVal :: ST a -> ((String,String),a)
getVal (ST p) = (p undefined)
```

which passes some arbitrary state `undefined` to the transformation returned by the do-expression.

So evaluation (using $\rightsquigarrow$ for "reduces to") of t_0 reads

$$
\begin{aligned}
& t_0 \\
\rightsquigarrow\ & \texttt{getVal}\ \mathrm{do}\ \{\ \texttt{initialise}\ \} \\
\rightsquigarrow\ & \texttt{getVal initialise} \\
\rightsquigarrow\ & \texttt{getVal}\ (\_ \mapsto ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ), ())) \\
\rightsquigarrow\ & (\_ \mapsto ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ), ()))\ \texttt{undefined} \\
\rightsquigarrow\ & ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ), ())
\end{aligned}
$$

✎ **Second example** Now we start to fill the skeleton given above. Let us just output one character:

```
t1 = getVal (do initialise
                putc 'c'
            )
```

After initialisation, this program writes `c` to the output stream. Setting $t :=$ `initialise` and $f :=$ `putc 'c'`, the evaluation of the program body (i.e., the do-notation) reads

$$
\begin{aligned}
& \mathrm{do}\ \{\ t;\ f\ \} \\
\rightsquigarrow\ & t \gg\!= (\_ \mapsto \mathrm{do}\ \{\ f\ \}) \\
\rightsquigarrow\ & t \gg\!= (\_ \mapsto f) \\
\rightsquigarrow\ & \quad [\ \text{definition of } \gg\!= \text{ using Notation 6.5.2} \\
& s \mapsto ((\_ \mapsto f)[ts]_2)[ts]_1 \\
\rightsquigarrow\ & s \mapsto f[ts]_1 \\
\rightsquigarrow\ & \quad [\ t \text{ always maps to the initial state } ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ),()) \\
& s \mapsto f(\ \texttt{"foo"}\ ,\ \texttt{""}\ ) \\
\rightsquigarrow\ & \quad [\ f = \texttt{putc 'c'} \\
& s \mapsto ((\ \texttt{"foo"}\ ,\ \texttt{"c"}\ ), ())
\end{aligned}
$$

So, the do-notation returns a transformation that transforms an arbitrary initial state s into the state (`"foo"` , `"c"`) and the return value $()$. The `getVal` function binds s to `undefined`, and the *state of the world* after evaluation of the program is returned.

✎ **Third example** This program prints the first input character to the output stream

```
t2 = getVal (do initialise
                x <- getc
                putc x
             )
```

Again, setting $t :=$ `initialise` , $f :=$ `getc` , $g :=$ `putc` and $s_0 :=$ ((`"foo"` , `""`), ()), the evaluation of the do-expression reads

$$
\begin{array}{ll}
 & \text{do } \{\ t;\ x \leftarrow f;\ gx\ \} \\
\rightsquigarrow & \\
 & t \gg\!= (\_ \mapsto f \gg\!= (x \mapsto gx)) \\
\rightsquigarrow & \\
 & t \gg\!= (\_ \mapsto f \gg\!= g) \\
\rightsquigarrow & \\
 & s \mapsto ((\_ \mapsto f \gg\!= g)[ts]_2)[ts]_1 \\
\rightsquigarrow & \quad [\text{ since the } \lambda\text{-expression ignores its argument} \\
 & s \mapsto (f \gg\!= g))[ts]_1 \\
\rightsquigarrow & \quad [\text{ using the definition of } t \text{ which also ignores its input } s \\
 & \_ \mapsto (f \gg\!= g)\ s_0 \\
\rightsquigarrow & \\
 & \_ \mapsto (\quad s \mapsto (g[fs]_2)[fs]_1 \quad)\ s_0 \\
\rightsquigarrow & \\
 & \_ \mapsto (g[fs_0]_2)[fs_0]_1 \\
\rightsquigarrow & \quad [\text{ using } fs_0 = ((\texttt{"oo"}, \texttt{""}), \texttt{'f'}) \\
 & \_ \mapsto (g\ \texttt{'f'})\ (\texttt{"oo"}, \texttt{""}) \\
\rightsquigarrow & \\
 & \_ \mapsto ((\texttt{"oo"}, \texttt{"f"}), ())\ .
\end{array}
$$

✎ **Fourth example** Since the do-notation works on a list of transformations, and also returns a transformation, we can nest them. This leads to the ability to model *subroutines* in an imperative style.

We define a (recursive) function that prints a string:

```
puts :: String -> ST ()
puts "" = return ()
puts (x:xs) = do putc x
                 puts xs  --recursion
```

and then use it in

```
t3 = getVal (do initialise
                getc
                x <- getc
                puts "2nd char is "  --call puts
                putc x
             )
```

It should be clear how these functions are evaluated from the examples above. Therefore we do not go through this now.

6.6 The IO monad

A closer look at the definition of the ST monad reveals, that the pair of strings we used to model input and output streams is not mentioned at all. In fact, the definition of ST is a well known method to pass around state information between function calls.

One might argue that these achievements are vile, providing nothing more than syntactic sugar that avoids augmentation of function signatures to explicitly pass a state.

However, this is not entirely true.

The big advantage is, that the monad structure may be some kind of "one-way" object. Neither Definition 6.1.1, nor the definition given in Theorem 6.3.5, require a method to get things back out of the monad. In the last section, we used a function `getVal` to reveal the innards of the monad, but none of the laws and theorems mentioned above guarantees the existence of such a function. The same applies to the function `initialise`.

Recall, that we accepted the structure of objects in a category to be unknown (Definition 3.1.1). If, for example, we define a monad with a functor M, and apply M to an object A with well known structure in our category, we end up with another object MA whose structure might be unknown.

And this is all there is about the IO monad. The Haskell programmer has no clue about its internal structure, and it is not possible to access the *state of the world*. This protects one from doing funny things, like, e.g., make a copy of the *state of the world* and transform both "versions" of the world in different ways.

You may have noticed the similarity between the functions

```
getc :: ST Char
putc :: Char -> ST ()
puts :: String -> ST ()
```

defined above, and the standard IO functions

```
getChar :: IO Char
putChar :: Char -> IO ()
putStr :: String -> IO ()
```

provided by Haskell.

The main difference is that we do not know the internal structure of the IO monad. All we know is how to pass an object in that monad (MA) to a morphism ($f : A \to MB$) using the bind operator, thereby applying f to the "*value*" of the passed object.

Final remarks

This guide explored the theoretical backgrounds of Haskell's IO monad. On the way we have seen functors, natural transformations, and finally monads. All these purely theoretic concepts appeared to have a quite practical correspondence in the Haskell programming language, as emphasized by the according examples.

It should be clear now, how the IO monad is used to pass around the *state of the world*, without allowing the programmer to access it "too much".

Recall the novice Haskell programmer mentioned in the introduction:

> *"I'll just wrap the* `getLine` *in another function which only returns what I really need, maybe convert the users' input to an* `Int` *and return that."*

Now we can explain why the `IO` "thing" will always stick to a function that is somehow involved in doing IO. But with the knowledge about the bind operator, we can also talk "monad-free" functions into working on data returned from IO.

Also, we know why `IO ()` denotes the type of a "void IO procedure", and how the variable assignment notation `x<-...` is translated into λ abstraction.

Bibliography

[1] Maarten Fokkinga. *A Gentle Introduction to Category Theory — the calculational approach*. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, Part I, pages 1–72. University of Utrecht, Netherlands, September 1992.

[2] Michael Barr, Charles Wells. *Category Theory Lecture Notes for ESSLLI*. In *Lecture Notes for ESSLLI'99*, 1999. `http://www.let.uu.nl/esslli/Courses/barr/barrwells.ps`.

[3] Jeff Newbern. *All About Monads*. `http://www.nomaware.com/monads/html/index.html`.

[4] Theodore Norvell. *Monads for the Working Haskell Programmer — a short tutorial*. Memorial University of Newfoundland. `http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm`.

[5] Simon Peyton Jones, et. al. *Haskell 98 Language and Libraries — The Revised Report*. December 2002. `http://haskell.org/definition/`.

Monadic Parser Combinators

Graham Hutton
University of Nottingham

Erik Meijer
University of Utrecht

Appears as technical report NOTTCS-TR-96-4,
Department of Computer Science, University of Nottingham, 1996

Abstract

In functional programming, a popular approach to building recursive descent *parsers* is to model parsers as functions, and to define higher-order functions (or *combinators*) that implement grammar constructions such as sequencing, choice, and repetition. Such parsers form an instance of a *monad*, an algebraic structure from mathematics that has proved useful for addressing a number of computational problems. The purpose of this article is to provide a step-by-step tutorial on the monadic approach to building functional parsers, and to explain some of the benefits that result from exploiting monads. No prior knowledge of parser combinators or of monads is assumed. Indeed, this article can also be viewed as a first introduction to the use of monads in programming.

Contents

1 Introduction

In functional programming, a popular approach to building recursive descent *parsers* is to model parsers as functions, and to define higher-order functions (or *combinators*) that implement grammar constructions such as sequencing, choice, and repetition. The basic idea dates back to at least Burge's book on recursive programming techniques (Burge, 1975), and has been popularised in functional programming by Wadler (1985), Hutton (1992), Fokker (1995), and others. Combinators provide a quick and easy method of building functional parsers. Moreover, the method has the advantage over functional parser generators such as Ratatosk (Mogensen, 1993) and Happy (Gill & Marlow, 1995) that one has the full power of a functional language available to define new combinators for special applications (Landin, 1966).

It was realised early on (Wadler, 1990) that parsers form an instance of a *monad*, an algebraic structure from mathematics that has proved useful for addressing a number of computational problems (Moggi, 1989; Wadler, 1990; Wadler, 1992a; Wadler, 1992b). As well as being interesting from a mathematical point of view, recognising the monadic nature of parsers also brings practical benefits. For example, using a monadic sequencing combinator for parsers avoids the messy manipulation of nested tuples of results present in earlier work. Moreover, using *monad comprehension* notation makes parsers more compact and easier to read.

Taking the monadic approach further, the monad of parsers can be expressed in a modular way in terms of two simpler monads. The immediate benefit is that the basic parser combinators no longer need to be defined explicitly. Rather, they arise automatically as a special case of lifting monad operations from a base monad m to a certain other monad parameterised over m. This also means that, if we change the nature of parsers by modifying the base monad (for example, limiting parsers to producing at most one result), then new combinators for the modified monad of parsers also arise automatically via the lifting construction.

The purpose of this article is to provide a step-by-step tutorial on the monadic approach to building functional parsers, and to explain some of the benefits that result from exploiting monads. Much of the material is already known. Our contributions are the organisation of the material into a tutorial article; the introduction of new combinators for handling lexical issues without a separate lexer; and a new approach to implementing the offside rule, inspired by the use of monads.

Some prior exposure to functional programming would be helpful in reading this article, but special features of Gofer (Jones, 1995b) — our implementation language — are explained as they are used. Any other lazy functional language that supports (multi-parameter) constructor classes and the use of monad comprehension notation would do equally well. No prior knowledge of parser combinators or monads is assumed. Indeed, this article can also be viewed as a first introduction to the use of monads in programming. A library of monadic parser combinators taken from this article is available from the authors, via the World-Wide-Web.

2 Combinator parsers

We begin by reviewing the basic ideas of combinator parsing (Wadler, 1985; Hutton, 1992; Fokker, 1995). In particular, we define a type for parsers, three primitive parsers, and two primitive combinators for building larger parsers.

2.1 The type of parsers

Let us start by thinking of a parser as a function that takes a string of characters as input and yields some kind of tree as result, with the intention that the tree makes explicit the grammatical structure of the string:

```
type Parser = String -> Tree
```

In general, however, a parser might not consume all of its input string, so rather than the result of a parser being just a tree, we also return the unconsumed suffix of the input string. Thus we modify our type of parsers as follows:

```
type Parser = String -> (Tree,String)
```

Similarly, a parser might fail on its input string. Rather than just reporting a run-time error if this happens, we choose to have parsers return a list of pairs rather than a single pair, with the convention that the empty list denotes failure of a parser, and a singleton list denotes success:

```
type Parser = String -> [(Tree,String)]
```

Having an explicit representation of failure and returning the unconsumed part of the input string makes it possible to define combinators for building up parsers piecewise from smaller parsers. Returning a list of results opens up the possibility of returning more than one result if the input string can be parsed in more than one way, which may be the case if the underlying grammar is ambiguous.

Finally, different parsers will likely return different kinds of trees, so it is useful to abstract on the specific type `Tree` of trees, and make the type of result values into a parameter of the `Parser` type:

```
type Parser a = String -> [(a,String)]
```

This is the type of parsers we will use in the remainder of this article. One could go further (as in (Hutton, 1992), for example) and abstract upon the type `String` of tokens, but we do not have need for this generalisation here.

2.2 Primitive parsers

The three primitive parsers defined in this section are the building blocks of combinator parsing. The first parser is `result v`, which succeeds without consuming any of the input string, and returns the single result `v`:

```
result  :: a -> Parser a
result v = \inp -> [(v,inp)]
```

An expression of the form `\x -> e` is called a λ-abstraction, and denotes the function that takes an argument `x` and returns the value of the expression `e`. Thus `result v` is the function that takes an input string `inp` and returns the singleton list `[(v,inp)]`. This function could equally well be defined by `result v inp = [(v,inp)]`, but we prefer the above definition (in which the argument `inp` is shunted to the body of the definition) because it corresponds more closely to the type `result :: a -> Parser a`, which asserts that `result` is a function that takes a single argument and returns a parser.

Dually, the parser `zero` always fails, regardless of the input string:

```
zero :: Parser a
zero  = \inp -> []
```

Our final primitive is `item`, which successfully consumes the first character if the input string is non-empty, and fails otherwise:

```
item :: Parser Char
item  = \inp -> case inp of
                  []     -> []
                  (x:xs) -> [(x,xs)]
```

2.3 Parser combinators

The primitive parsers defined above are not very useful in themselves. In this section we consider how they can be glued together to form more useful parsers. We take our lead from the BNF notation for specifying grammars, in which larger grammars are built up piecewise from smaller grammars using a *sequencing* operator — denoted by juxtaposition — and a *choice* operator — denoted by a vertical bar |. We define corresponding operators for combining parsers, such that the structure of our parsers closely follows the structure of the underlying grammars.

In earlier (non-monadic) accounts of combinator parsing (Wadler, 1985; Hutton, 1992; Fokker, 1995), sequencing of parsers was usually captured by a combinator

```
seq      :: Parser a -> Parser b -> Parser (a,b)
p `seq` q = \inp -> [((v,w),inp'') | (v,inp')  <- p inp
                                   , (w,inp'') <- q inp']
```

that applies one parser after another, with the results from the two parsers being combined as pairs. The infix notation `p `seq` q` is syntactic sugar for `seq p q`; any function of two arguments can used as an infix operator in this way, by enclosing its name in backquotes. At first sight, the `seq` combinator might seem a natural composition primitive. In practice, however, using `seq` leads to parsers with nested tuples as results, which are messy to manipulate.

The problem of nested tuples can be avoided by adopting a *monadic* sequencing combinator (commonly known as `bind`) which integrates the sequencing of parsers with the processing of their result values:

```
bind      :: Parser a -> (a -> Parser b) -> Parser b
p `bind` f = \inp -> concat [f v inp' | (v,inp') <- p inp]
```

The definition for `bind` can be interpreted as follows. First of all, the parser `p` is applied to the input string, yielding a list of (value,string) pairs. Now since `f` is a function that takes a value and returns a parser, it can be applied to each value (and unconsumed input string) in turn. This results in a list of lists of (value,string) pairs, that can then be flattened to a single list using `concat`.

The `bind` combinator avoids the problem of nested tuples of results because the results of the first parser are made directly available for processing by the second, rather than being paired up with the other results to be processed later on. A typical parser built using `bind` has the following structure

```
p1 `bind` \x1 ->
p2 `bind` \x2 ->
...
pn `bind` \xn ->
result (f x1 x2 ... xn)
```

and can be read operationally as follows: apply parser `p1` and call its result value `x1`; then apply parser `p2` and call its result value `x2`; ...; then apply the parser `pn` and call its result value `xn`; and finally, combine all the results into a single value by applying the function `f`. For example, the `seq` combinator can be defined by

```
p `seq` q = p `bind` \x ->
            q `bind` \y ->
            result (x,y)
```

(On the other hand, `bind` cannot be defined in terms of `seq`.)

Using the `bind` combinator, we are now able to define some simple but useful parsers. Recall that the `item` parser consumes a single character unconditionally. In practice, we are normally only interested in consuming certain specific characters. For this reason, we use `item` to define a combinator `sat` that takes a predicate (a Boolean valued function), and yields a parser that consumes a single character if it satisfies the predicate, and fails otherwise:

```
sat  :: (Char -> Bool) -> Parser Char
sat p = item `bind` \x ->
        if p x then result x else zero
```

Note that if `item` fails (that is, if the input string is empty), then so does `sat p`, since it can readily be observed that `zero `bind` f = zero` for all functions `f` of the appropriate type. Indeed, this equation is not specific to parsers: it holds for an arbitrary *monad with a zero* (Wadler, 1992a; Wadler, 1992b). Monads and their connection to parsers will be discussed in the next section.

Using `sat`, we can define parsers for specific characters, single digits, lower-case letters, and upper-case letters:

```
char  :: Char -> Parser Char
char x = sat (\y -> x == y)
```

```
digit :: Parser Char
digit  = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower  = sat (\x -> 'a' <= x && x <= 'z')

upper :: Parser Char
upper  = sat (\x -> 'A' <= x && x <= 'Z')
```

For example, applying the parser `upper` to the input string `"Hello"` succeeds with the single successful result `[('H',"ello")]`, since the `upper` parser succeeds with `'H'` as the result value and `"ello"` as the unconsumed suffix of the input. On the other hand, applying the parser `lower` to the string `"Hello"` fails with `[]` as the result, since `'H'` is not a lower-case letter.

As another example of using `bind`, consider the parser that accepts two lower-case letters in sequence, returning a string of length two:

```
lower `bind` \x ->
lower `bind` \y ->
result [x,y]
```

Applying this parser to the string `"abcd"` succeeds with the result `[("ab","cd")]`. Applying the same parser to `"aBcd"` fails with the result `[]`, because even though the initial letter `'a'` can be consumed by the first `lower` parser, the following letter `'B'` cannot be consumed by the second `lower` parser.

Of course, the above parser for two letters in sequence can be generalised to a parser for arbitrary strings of lower-case letters. Since the length of the string to be parsed cannot be predicted in advance, such a parser will naturally be defined recursively, using a choice operator to decide between parsing a single letter and recursing, or parsing nothing further and terminating. A suitable choice combinator for parsers, `plus`, is defined as follows:

```
plus       :: Parser a -> Parser a -> Parser a
p `plus` q  = \inp -> (p inp ++ q inp)
```

That is, both argument parsers `p` and `q` are applied to the same input string, and their result lists are concatenated to form a single result list. Note that it is not required that `p` and `q` accept disjoint sets of strings: if both parsers succeed on the input string then more than one result value will be returned, reflecting the different ways that the input string can be parsed.

As examples of using `plus`, some of our earlier parsers can now be combined to give parsers for letters and alpha-numeric characters:

```
letter   :: Parser Char
letter    = lower `plus` upper

alphanum :: Parser Char
alphanum  = letter `plus` digit
```

More interestingly, a parser for words (strings of letters) is defined by

```
word :: Parser String
word  = neWord `plus` result ""
        where
           neWord = letter `bind` \x  ->
                    word   `bind` \xs ->
                    result (x:xs)
```

That is, `word` either parses a non-empty word (a single letter followed by a word, using a recursive call to `word`), in which case the two results are combined to form a string, or parses nothing and returns the empty string.

For example, applying `word` to the input `"Yes!"` gives the result `[("Yes","!"), ("Ye","s!"), ("Y","es!"), ("","Yes!")]`. The first result, `("Yes","!")`, is the expected result: the string of letters `"Yes"` has been consumed, and the unconsumed input is `"!"`. In the subsequent results a decreasing number of letters are consumed. This behaviour arises because the choice operator `plus` is *non-deterministic*: both alternatives can be explored, even if the first alternative is successful. Thus, at each application of `letter`, there is always the option to just finish parsing, even if there are still letters left to be consumed from the start of the input.

3 Parsers and monads

Later on we will define a number of useful parser combinators in terms of the primitive parsers and combinators just defined. But first we turn our attention to the monadic nature of combinator parsers.

3.1 The parser monad

So far, we have defined (among others) the following two operations on parsers:

```
result :: a -> Parser a
bind   :: Parser a -> (a -> Parser b) -> Parser b
```

Generalising from the specific case of `Parser` to some arbitrary type constructor `M` gives the notion of a monad: a *monad* is a type constructor `M` (a function from types to types), together with operations `result` and `bind` of the following types:

```
result :: a -> M a
bind   :: M a -> (a -> M b) -> M b
```

Thus, parsers form a monad for which `M` is the `Parser` type constructor, and `result` and `bind` are defined as previously. Technically, the two operations of a monad must also satisfy a few algebraic properties, but we do not concern ourselves with such properties here; see (Wadler, 1992a; Wadler, 1992b) for more details.

Readers familiar with the categorical definition of a monad may have expected two operations `map :: (a -> b) -> (M a -> M b)` and `join :: M (M a) -> M a` in place of the single operation `bind`. However, our definition is equivalent to the

categorical one (Wadler, 1992a; Wadler, 1992b), and has the advantage that `bind` generally proves more convenient for monadic programming than `map` and `join`.

Parsers are not the only example of a monad. Indeed, we will see later on how the parser monad can be re-formulated in terms of two simpler monads. This raises the question of what to do about the naming of the monadic combinators `result` and `bind`. In functional languages based upon the Hindley-Milner typing system (for example, Miranda[1] and Standard ML) it is not possible to use the same names for the combinators of different monads. Rather, one would have to use different names, such as `resultM` and `bindM`, for the combinators of each monad `M`.

Gofer, however, extends the Hindley-Milner typing system with an overloading mechanism that permits the use of the same names for the combinators of different monads. Under this overloading mechanism, the appropriate monad for each use of a name is calculated automatically during type inference.

Overloading in Gofer is accomplished by the use of *classes* (Jones, 1995c). A class for monads can be declared in Gofer by:

```
class Monad m where
   result :: a -> m a
   bind   :: m a -> (a -> m b) -> m b
```

This declaration can be read as follows: a type constructor `m` is a member of the class `Monad` if it is equipped with `result` and `bind` operations of the specified types. The fact that `m` must be a type constructor (rather than just a type) is inferred from its use in the types for the operations.

Now the type constructor `Parser` can be made into an instance of the class `Monad` using the `result` and `bind` from the previous section:

```
instance Monad Parser where
   -- result :: a -> Parser a
   result v   = \inp -> [(v,inp)]

   -- bind   :: Parser a -> (a -> Parser b) -> Parser b
   p `bind` f = \inp -> concat [f v out | (v,out) <- p inp]
```

We pause briefly here to address a couple of technical points concerning Gofer. First of all, type synonyms such as `Parser` must be supplied with all their arguments. Hence the instance declaration above is not actually valid Gofer code, since `Parser` is used in the first line without an argument. The problem is easy to solve (redefine `Parser` using `data` rather than `type`, or as a *restricted* type synonym), but for simplicity we prefer in this article just to assume that type synonyms *can* be partially applied. The second point is that the syntax of Gofer does not currently allow the types of the defined functions in instance declarations to be explicitly specified. But for clarity, as above, we include such types in comments.

Let us turn now to the following operations on parsers:

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

```
zero :: Parser a
plus :: Parser a -> Parser a -> Parser a
```

Generalising once again from the specific case of the `Parser` type constructor, we arrive at the notion of a *monad with a zero and a plus*, which can be encapsulated using the Gofer class system in the following manner:

```
class Monad m => MonadOPlus m where
   zero :: m a
   (++) :: m a -> m a -> m a
```

That is, a type constructor `m` is a member of the class `MonadOPlus` if it is a member of the class `Monad` (that is, it is equipped with a `result` and `bind`), and if it is also equipped with `zero` and `(++)` operators of the specified types. Of course, the two extra operations must also satisfy some algebraic properties; these are discussed in (Wadler, 1992a; Wadler, 1992b). Note also that `(++)` is used above rather than `plus`, following the example of lists: we will see later on that lists form a monad for which the plus operation is just the familiar append operation `(++)`.

Now since `Parser` is already a monad, it can be made into a monad with a zero and a plus using the following definitions:

```
instance MonadOPlus Parser where
   -- zero :: Parser a
   zero     = \inp -> []

   -- (++) :: Parser a -> Parser a -> Parser a
   p ++ q   = \inp -> (p inp ++ q inp)
```

3.2 Monad comprehension syntax

So far we have seen one advantage of recognising the monadic nature of parsers: the monadic sequencing combinator `bind` handles result values better than the conventional sequencing combinator `seq`. In this section we consider another advantage of the monadic approach, namely that *monad comprehension* syntax can be used to make parsers more compact and easier to read.

As mentioned earlier, many parsers will have a structure as a sequence of `bind`s followed by single call to `result`:

```
p1 `bind` \x1 ->
p2 `bind` \x2 ->
...
pn `bind` \xn ->
result (f x1 x2 ... xn)
```

Gofer provides a special notation for defining parsers of this shape, allowing them to be expressed in the following, more appealing form:

```
[ f x1 x2 ... xn | x1 <- p1
```

```
        , x2 <- p2
        , ...
        , xn <- pn ]
```

In fact, this notation is not specific to parsers, but can be used with any monad (Jones, 1995c). The reader might notice the similarity to the list comprehension notation supported by many functional languages. It was Wadler (1990) who first observed that the comprehension notation is not particular to lists, but makes sense for an arbitrary monad. Indeed, the algebraic properties required of the monad operations turn out to be precisely those required for the notation to make sense. To our knowledge, Gofer is the first language to implement Wadler's *monad comprehension* notation. Using this notation can make parsers much easier to read, and we will use the notation in the remainder of this article.

As our first example of using comprehension notation, we define a parser for recognising specific strings, with the string itself returned as the result:

```
string       :: String -> Parser String
string ""     = [""]
string (x:xs) = [x:xs | _ <- char x, _ <- string xs]
```

That is, if the string to be parsed is empty we just return the empty string as the result; `[""]` is just monad comprehension syntax for `result ""`. Otherwise, we parse the first character of the string using `char`, and then parse the remaining characters using a recursive call to `string`. Without the aid of comprehension notation, the above definition would read as follows:

```
string       :: String -> Parser String
string ""     = result ""
string (x:xs) = char x    `bind` \_ ->
                string xs `bind` \_ ->
                result (x:xs)
```

Note that the parser `string xs` fails if only a prefix of the given string `xs` is recognised in the input. For example, applying the parser `string "hello"` to the input `"hello there"` gives the successful result `[("hello"," there")]`. On the other hand, applying the same parser to `"helicopter"` fails with the result `[]`, even though the prefix `"hel"` of the input can be recognised.

In list comprehension notation, we are not just restricted to *generators* that bind variables to values, but can also use Boolean-valued *guards* that restrict the values of the bound variables. For example, a function `negs` that selects all the negative numbers from a list of integers can be expressed as follows:

```
negs   :: [Int] -> [Int]
negs xs = [x | x <- xs, x < 0]
```

In this case, the expression `x < 0` is a guard that restricts the variable `x` (bound by the generator `x <- xs`) to only take on values less than zero.

Wadler (1990) observed that the use of guards makes sense for an arbitrary

monad with a zero. The monad comprehension notation in Gofer supports this use of guards. For example, the `sat` combinator

```
sat  :: (Char -> Bool) -> Parser Char
sat p = item `bind` \x ->
        if p x then result x else zero
```

can be defined more succinctly using a comprehension with a guard:

```
sat  :: (Char -> Bool) -> Parser Char
sat p = [x | x <- item, p x]
```

We conclude this section by noting that there is another notation that can be used to make monadic programs easier to read: the so-called "`do`" notation (Jones, 1994; Jones & Launchbury, 1994). For example, using this notation the combinators `string` and `sat` can be defined as follows:

```
string       :: String -> Parser String
string ""     = do { result "" }
string (x:xs) = do { char x ; string xs ; result (x:xs) }

sat          :: (Char -> Bool) -> Parser Char
sat p         = do { x <- item ; if (p x) ; result x }
```

The `do` notation has a couple of advantages over monad comprehension notation: we are not restricted to monad expressions that end with a use of `result`; and generators of the form `_ <- e` that do not bind variables can be abbreviated by `e`. The `do` notation is supported by Gofer, but monad expressions involving parsers typically end with a use of `result` (to compute the result value from the parser), so the extra generality is not really necessary in this case. For this reason, and for simplicity, in this article we only use the comprehension notation. It would be an easy task, however, to translate our definitions into the `do` notation.

4 Combinators for repetition

Parser generators such as Lex and Yacc (Aho *et al.*, 1986) for producing parsers written in C, and Ratatosk (Mogensen, 1993) and Happy (Gill & Marlow, 1995) for producing parsers written in Haskell, typically offer a fixed set of combinators for describing grammars. In contrast, with the method of building parsers as presented in this article the set of combinators is completely extensible: parsers are first-class values, and we have the full power of a functional language at our disposal to define special combinators for special applications.

In this section we define combinators for a number of common patterns of *repetition*. These combinators are not specific to parsers, but can be used with an arbitrary monad with a zero and plus. For clarity, however, we specialise the types of the combinators to the case of parsers.

In subsequent sections we will introduce combinators for other purposes, including handling lexical issues and Gofer's offside rule.

4.1 Simple repetition

Earlier we defined a parser `word` for consuming zero or more letters from the input string. Using monad comprehension notation, the definition is:

```
word :: Parser String
word  = [x:xs | x <- letter, xs <- word] ++ [""]
```

We can easily imagine a number of other parsers that exhibit a similar structure to `word`. For example, parsers for strings of digits or strings of spaces could be defined in precisely the same way, the only difference being be that the component parser `letter` would be replaced by either `digit` or `char ' '`. To avoid defining a number of different parsers with a similar structure, we abstract on the pattern of recursion in `word` and define a general combinator, `many`, that parses sequences of items.

The combinator `many` applies a parser `p` zero or more times to an input string. The results from each application of `p` are returned in a list:

```
many  :: Parser a -> Parser [a]
many p = [x:xs | x <- p, xs <- many p] ++ [[]]
```

Different parsers can be made by supplying different arguments parsers `p`. For example, `word` can be defined just as `many letter`, and the other parsers mentioned above by `many digit` and `many (char ' ')`.

Just as the original `word` parser returns many results in general (decreasing in the number of letters consumed from the input), so does `many p`. Of course, in most cases we will only be interested in the first parse from `many p`, in which `p` is successfully applied as many times as possible. We will return to this point in the next section, when we address the efficiency of parsers.

As another application of `many`, we can define a parser for identifiers. For simplicity, we regard an identifier as a lower-case letter followed by zero or more alphanumeric characters. It would be easy to extend the definition to handle extra characters, such as underlines or backquotes.

```
ident :: Parser String
ident  = [x:xs | x <- lower, xs <- many alphanum]
```

Sometimes we will only be interested in non-empty sequences of items. For this reason we define a special combinator, `many1`, in terms of `many`:

```
many1  :: Parser a -> Parser [a]
many1 p = [x:xs | x <- p, xs <- many p]
```

For example, applying `many1 (char 'a')` to the input `"aaab"` gives the result `[("aaa","b"), ("aa","ab"), ("a","aab")]`, which is the same as for `many (char 'a')`, except that the final pair `("", "aaab")` is no longer present. Note also that `many1 p` may fail, whereas `many p` always succeeds.

Using `many1` we can define a parser for natural numbers:

```
nat :: Parser Int
nat  = [eval xs | xs <- many1 digit]
```

```
      where
         eval xs  = foldl1 op [ord x - ord '0' | x <- xs]
         m `op` n = 10*m + n
```

In turn, `nat` can be used to define a parser for integers:

```
int :: Parser Int
int  = [-n | _ <- char '-', n <- nat] ++ nat
```

A more sophisticated way to define `int` is as follows. First try and parse the negation character '-'. If this is successful then return the negation function as the result of the parse; otherwise return the identity function. The final step is then to parse a natural number, and use the function returned by attempting to parse the '-' character to modify the resulting number:

```
int :: Parser Int
int  = [f n | f <- op, n <- nat]
       where
          op = [negate | _ <- char '-'] ++ [id]
```

4.2 Repetition with separators

The `many` combinators parse sequences of items. Now we consider a slightly more general pattern of repetition, in which separators between the items are involved. Consider the problem of parsing a non-empty list of integers, such as `[1,-42,17]`. Such a parser can be defined in terms of the `many` combinator as follows:

```
ints :: Parser [Int]
ints  = [n:ns | _  <- char '['
              , n  <- int
              , ns <- many [x | _ <- char ',', x <- int]
              , _  <- char ']']
```

As was the case in the previous section for the `word` parser, we can imagine a number of other parsers with a similar structure to `ints`, so it is useful to abstract on the pattern of repetition and define a general purpose combinator, which we call `sepby1`. The combinator `sepby1` is like `many1` in that it recognises non-empty sequences of a given parser `p`, but different in that the instances of `p` are separated by a parser `sep` whose result values are ignored:

```
sepby1        :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = [x:xs | x  <- p
                       , xs <- many [y | _ <- sep, y <- p]]
```

Note that the fact that the results of the `sep` parser are ignored is reflected in the type of the `sepby1` combinator: the `sep` parser gives results of type `b`, but this type does not occur in the type `[a]` of the results of the combinator.

Now `ints` can be defined in a more compact form:

```
ints = [ns | _  <- char '['
           , ns <- int `sepby1` char ','
           , _  <- char ']']
```

In fact we can go a little further. The bracketing of parsers by other parsers whose results are ignored — in the case above, the bracketing parsers are `char '['` and `char ']'` — is common enough to also merit its own combinator:

```
bracket :: Parser a -> Parser b -> Parser c -> Parser b
bracket open p close = [x | _ <- open, x <- p, _ <- close]
```

Now `ints` can be defined just as

```
ints = bracket (char '[')
               (int `sepby1` char ',')
               (char ']')
```

Finally, while `many1` was defined in terms of `many`, the combinator `sepby` (for possibly-empty sequences) is naturally defined in terms of `sepby1`:

```
sepby         :: Parser a -> Parser b -> Parser [a]
p `sepby` sep  = (p `sepby1` sep) ++ [[]]
```

4.3 Repetition with meaningful separators

The `sepby` combinators handle the case of parsing sequences of items separated by text that can be ignored. In this final section on repetition, we address the more general case in which the separators themselves carry meaning. The combinators defined in this section are due to Fokker (1995).

Consider the problem of parsing simple arithmetic expressions such as `1+2-(3+4)`, built up from natural numbers using addition, subtraction, and parentheses. The two arithmetic operators are assumed to associate to the left (thus, for example, `1-2-3` should be parsed as `(1-2)-3`), and have the same precedence. The standard BNF grammar for such expressions is written as follows:

expr ::= *expr addop factor* | *factor*
addop ::= `+` | `-`
factor ::= *nat* | `(` *expr* `)`

This grammar can be translated directly into a combinator parser:

```
expr   :: Parser Int
addop  :: Parser (Int -> Int -> Int)
factor :: Parser Int

expr   = [f x y | x <- expr, f <- addop, y <- factor] ++ factor

addop  = [(+) | _ <- char '+'] ++ [(-) | _ <- char '-']

factor = nat ++ bracket (char '(') expr (char ')')
```

In fact, rather than just returning some kind of parse tree, the `expr` parser above actually evaluates arithmetic expressions to their integer value: the `addop` parser returns a function as its result value, which is used to combine the result values produced by parsing the arguments to the operator.

Of course, however, there is a problem with the `expr` parser as defined above. The fact that the operators associate to the left is taken account of by `expr` being *left-recursive* (the first thing it does is make a recursive call to itself). Thus `expr` never makes any progress, and hence does not terminate.

As is well-known, this kind of non-termination for parsers can be solved by replacing left-recursion by iteration. Looking at the *expr* grammar, we see that an expression is a sequence of *factor*s, separated by *addop*s. Thus the parser for expressions can be re-defined using `many` as follows:

```
expr = [... | x   <- factor
            , fys <- many [(f,y) | f <- addop, y <- factor]]
```

This takes care of the non-termination, but it still remains to fill in the "`...`" part of the new definition, which computes the value of an expression.

Suppose now that the input string is `"1-2+3-4"`. Then after parsing using `expr`, the variable `x` will be `1` and `fys` will be the list `[((-),2), ((+),3), ((-),4)]`. These can be reduced to a single value `1-2+3-4 = ((1-2)+3)-4 = -2` by folding: the built-in function `foldl` is such that, for example, `foldl g a [b,c,d,e] = ((a ‘g‘ b) ‘g‘ c) ‘g‘ d) ‘g‘ e`. In the present case, we need to take `g` as the function `\x (f,y) -> f x y`, and `a` as the integer `x`:

```
expr = [foldl (\x (f,y) -> f x y) x fys
          | x   <- factor
          , fys <- many [(f,y) | f <- addop, y <- factor]]
```

Now, for example, applying `expr` to the input string `"1+2-(3+4)"` gives the result `[(-4,""), (3,"-(3+4)", (1,"+2-(3+4)")]`, as expected.

Playing the generalisation game once again, we can abstract on the pattern of repetition in `expr` and define a new combinator. The combinator, `chainl1`, parses non-empty sequences of items separated by operators that associate to the left:

```
chainl1       :: Parser a -> Parser (a -> a -> a) -> Parser a
p ‘chainl1‘ op = [foldl (\x (f,y) -> f x y) x fys
                    | x   <- p
                    , fys <- many [(f,y) | f <- op, y <- p]]
```

Thus our parser for expressions can now be written as follows:

```
expr   = factor ‘chainl1‘ addop

addop  = [(+) | _ <- char '+'] ++ [(-) | _ <- char '-']

factor = nat ++ bracket (char '(') expr (char ')')
```

Most operator parsers will have a similar structure to `addop` above, so it is useful to abstract a combinator for building such parsers:

```
ops   :: [(Parser a, b)] -> Parser b
ops xs = foldr1 (++) [[op | _ <- p] | (p,op) <- xs]
```

The built-in function `foldr1` is such that, for example, `foldr1 g [a,b,c,d] = a ‘g‘ (b ‘g‘ (c ‘g‘ d))`. It is defined for any non-empty list. In the above case then, `foldr1` places the choice operator `(++)` between each parser in the list. Using `ops`, our `addop` parser can now be defined by

```
addop = ops [(char '+', (+)), (char '-', (-))]
```

A possible inefficiency in the definition of the `chainl1` combinator is the construction of the intermediate list `fys`. This can be avoided by giving a direct recursive definition of `chainl1` that does not make use of `foldl` and `many`, using an accumulating parameter to construct the final result:

```
chainl1       :: Parser a -> Parser (a -> a -> a) -> Parser a
p ‘chainl1‘ op = p ‘bind‘ rest
                 where
                    rest x = (op ‘bind‘ \f ->
                              p  ‘bind‘ \y ->
                              rest (f x y)) ++ [x]
```

This definition has a natural operational reading. The parser `p ‘chainl1‘ op` first parses a single `p`, whose result value becomes the initial accumulator for the `rest` function. Then it attempts to parse an operator and a single `p`. If successful, the accumulator and the result from `p` are combined using the function `f` returned from parsing the operator, and the resulting value becomes the new accumulator when parsing the remainder of the sequence (using a recursive call to `rest`). Otherwise, the sequence is finished, and the accumulator is returned.

As another interesting application of `chainl1`, we can redefine our earlier parser `nat` for natural numbers such that it does not construct an intermediate list of digits. In this case, the `op` parser does not do any parsing, but returns the function that combines a natural and a digit:

```
nat :: Parser Int
nat  = [ord x - ord '0' | x <- digit] ‘chainl1‘ [op]
       where
          m ‘op‘ n = 10*m + n
```

Naturally, we can also define a combinator `chainr1` that parses non-empty sequences of items separated by operators that associate to the *right*, rather than to the left. For simplicity, we only give the direct recursive definition:

```
chainr1       :: Parser a -> Parser (a -> a -> a) -> Parser a
p ‘chainr1‘ op =
   p ‘bind‘ \x ->
      [f x y | f <- op, y <- p ‘chainr1‘ op] ++ [x]
```

That is, `p ‘chainr1‘ op` first parses a single `p`. Then it attempts to parse an operator and the rest of the sequence (using a recursive call to `chainr1`). If successful,

the pair of results from the first `p` and the rest of the sequence are combined using the function `f` returned from parsing the operator. Otherwise, the sequence is finished, and the result from `p` is returned.

As an example of using `chainr1`, we extend our parser for arithmetic expressions to handle exponentiation; this operator has higher precedence than the previous two operators, and associates to the right:

```
expr   = term   `chainl1` addop

term   = factor `chainr1` expop

factor = nat ++ bracket (char '(') expr (char ')')

addop  = ops [(char '+', (+)), (char '-', (-))]

expop  = ops [(char '^', (^))]
```

For completeness, we also define combinators `chainl` and `chainr` that have the same behaviour as `chainl1` and `chainr1`, except that they can also consume no input, in which case a given value `v` is returned as the result:

```
chainl :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op v = (p `chainl1` op) ++ [v]

chainr :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainr p op v = (p `chainr1` op) ++ [v]
```

In summary then, `chainl` and `chainr` provide a simple way to build parsers for expression-like grammars. Using these combinators avoids the need for transformations to remove left-recursion in the grammar, that would otherwise result in non-termination of the parser. They also avoid the need for left-factorisation of the grammar, that would otherwise result in unnecessary backtracking; we will return to this point in the next section.

5 Efficiency of parsers

Using combinators is a simple and flexible method of building parsers. However, the power of the combinators — in particular, their ability to backtrack and return multiple results — can lead to parsers with unexpected space and time performance if one does not take care. In this section we outline some simple techniques that can be used to improve the efficiency of parsers. Readers interested in further techniques are referred to Röjemo's thesis (1995), which contains a chapter on the use of heap profiling tools in the optimisation of parser combinators.

5.1 Left factoring

Consider the simple problem of parsing and evaluating two natural numbers separated by the addition symbol '`+`', or by the subtraction symbol '`-`'. This specification can be translated directly into the following parser:

```
eval :: Parser Int
eval  = add ++ sub
        where
           add = [x+y | x <- nat, _ <- char '+', y <- nat]
           sub = [x-y | x <- nat, _ <- char '-', y <- nat]
```

This parser gives the correct results, but is inefficient. For example, when parsing the string `"123-456"` the number `123` will first be parsed by the `add` parser, that will then fail because there is no '`+`' symbol following the number. The correct parse will only be found by backtracking in the input string, and parsing the number `123` *again*, this time from within the `sub` parser.

Of course, the way to avoid the possibility of backtracking and repeated parsing is to *left factorise* the `eval` parser. That is, the initial use of `nat` in the component parsers `add` and `sub` should be factorised out:

```
eval = [v | x <- nat, v <- add x ++ sub x]
       where
          add x = [x+y | _ <- char '+', y <- nat]
          sub x = [x+y | _ <- char '-', y <- nat]
```

This new version of `eval` gives the same results as the original version, but requires no backtracking. Using the new `eval`, the string `"123-456"` can now be parsed in linear time. In fact we can go a little further, and right factorise the remaining use of `nat` in both `add` and `sub`. This does not improve the efficiency of `eval`, but arguably gives a cleaner parser:

```
eval = [f x y | x <- nat
              , f <- ops [(char '+', (+)), (char '-', (-))]
              , y <- nat]
```

In practice, most cases where left factorisation of a parser is necessary to improve efficiency will concern parsers for some kind of expression. In such cases, manually factorising the parser will not be required, since expression-like parsers can be built using the `chain` combinators from the previous section, which already encapsulate the necessary left factorisation.

The motto of this section is the following: backtracking is a powerful tool, but it should not be used as a substitute for care in designing parsers.

5.2 Improving laziness

Recall the definition of the repetition combinator `many`:

```
many  :: Parser a -> Parser [a]
many p = [x:xs | x <- p, xs <- many p] ++ [[]]
```

For example, applying `many (char 'a')` to the input `"aaab"` gives the result `[("aaa","b"), ("aa","ab"), ("a","aab"),("","aaab")]`. Since Gofer is lazy, we would expect the `a`'s in the first result `"aaa"` to become available one at a time, as they are consumed from the input. This is not in fact what happens. In practice no part of the result `"aaa"` will be produced until all the `a`'s have been consumed. In other words, `many` is not as lazy as we would expect.

But does this really matter? Yes, because it is common in functional programming to rely on laziness to avoid the creation of large intermediate structures (Hughes, 1989). As noted by Wadler (1985; 1992b), what is needed to solve the problem with `many` is a means to make explicit that the parser `many p` always succeeds. (Even if `p` itself always fails, `many p` will still succeed, with the empty list as the result value.) This is the purpose of the `force` combinator:

```
force  :: Parser a -> Parser a
force p = \inp -> let x = p inp in
                  (fst (head x), snd (head x)) : tail x
```

Given a parser `p` that always succeeds, the parser `force p` has the same behaviour as `p`, except that before any parsing of the input string is attempted the result of the parser is immediately forced to take on the form $(\bot,\bot):\bot$, where $\bot$ represents a presently undefined value.

Using `force`, the `many` combinator can be re-defined as follows:

```
many  :: Parser a -> Parser [a]
many p = force ([x:xs | x <- p, xs <- many p] ++ [[]])
```

The use of `force` ensures that `many p` and all of its recursive calls return at least one result. The new definition of `many` now has the expected behaviour under lazy evaluation. For example, applying `many (char 'a')` to the partially-defined string $\texttt{'a'}:\bot$ gives the partially-defined result $(\texttt{'a'}:\bot,\bot):\bot$. In contrast, with the old version of `many`, the result for this example is the completely undefined value $\bot$.

Some readers might wonder why `force` is defined using the following selection functions, rather than by pattern matching?

```
fst :: (a,b) -> a      head :: [a] -> a
snd :: (a,b) -> b      tail :: [a] -> [a]
```

The answer is that, depending on the semantics of patterns in the particular implementation language, a definition of `force` using patterns might not have the expected behaviour under lazy evaluation.

5.3 Limiting the number of results

Consider the simple problem of parsing a natural number, or if no such number is present just returning the number 0 as the default result. A first approximation to such a parser might be as follows:

```
number :: Parser Int
number  = nat ++ [0]
```

However, this does not quite have the required behaviour. For example, applying `number` to the input `"hello"` gives the correct result `[(0,"hello")]`. On the other hand, applying `number` to `"123"` gives the result `[(123,""), (0,"123")]`, whereas we only really want the single result `[(123,"")]`.

One solution to the above problem is to make use of *deterministic* parser combinators (see section 7.5) — all parsers built using such combinators are restricted by construction to producing at most one result. A more general solution, however, is to retain the flexibility of the non-deterministic combinators, but to provide a means to make explicit that we are only interested in the first result produced by certain parsers, such as `number`. This is the purpose of the `first` combinator:

```
first  :: Parser a -> Parser a
first p = \inp -> case p inp of
                    []     -> []
                    (x:xs) -> [x]
```

Given a parser `p`, the parser `first p` has the same behaviour as `p`, except that only the first result (if any) is returned. Using `first` we can define a deterministic version `(+++)` of the standard choice combinator `(++)` for parsers:

```
(+++)  :: Parser a -> Parser a -> Parser a
p +++ q = first (p ++ q)
```

Replacing `(++)` by `(+++)` in `number` gives the desired behaviour.

As well as being used to ensure the correct behaviour of parsers, using `(+++)` can also improve their efficiency. As an example, consider a parser that accepts either of the strings `"yellow"` or `"orange"`:

```
colour :: Parser String
colour  = p1 ++ p2
          where
             p1 = string "yellow"
             p2 = string "orange"
```

Recall now the behaviour of the choice combinator `(++)`: it takes a string, applies both argument parsers to this string, and concatenates the resulting lists. Thus in the `colour` example, if `p1` is successfully applied then `p2` will still be applied to the same string, even though it is guaranteed to fail. This inefficiency can be avoided using `(+++)`, which ensures that if `p1` succeeds then `p2` is never applied:

```
colour = p1 +++ p2
         where
            p1 = string "yellow"
            p2 = string "orange"
```

More generally, if we know that a parser of the form `p ++ q` is deterministic (only ever returns at most one result value), then `p +++ q` has the same behaviour, but is more efficient: if `p` succeeds then `q` is never applied. In the remainder of this article it will mostly be the `(+++)` choice combinator that is used. For reasons of efficiency,

in the combinator libraries that accompany this article, the repetition combinators from the previous section are defined using `(+++)` rather than `(++)`.

We conclude this section by asking why `first` is defined by pattern matching, rather than by using the selection function `take :: Int -> [a] -> [a]` (where, for example, `take 3 "parsing" = "par"`):

```
first p = \inp -> take 1 (p inp)
```

The answer concerns the behaviour under lazy evaluation. To see the problem, let us unfold the use of `take` in the above definition:

```
first p = \inp -> case p inp of
                    []     -> []
                    (x:xs) -> x : take 0 xs
```

When the sub-expression `take 0 xs` is evaluated, it will yield `[]`. However, under lazy evaluation this computation will be suspended until its value is required. The effect is that the list `xs` may be retained in memory for some time, when in fact it can safely be discarded immediately. This is an example of a *space leak*. The definition of `first` using pattern matching does not suffer from this problem.

6 Handling lexical issues

Traditionally, a string to be parsed is not supplied directly to a parser, but is first passed through a lexical analysis phase (or *lexer*) that breaks the string into a sequence of tokens (Aho *et al.*, 1986). Lexical analysis is a convenient place to remove white-space (spaces, newlines, and tabs) and comments from the input string, and to distinguish between identifiers and keywords.

Since lexers are just simple parsers, they can be built using parser combinators, as discussed by Hutton (1992). However, as we shall see in this section, the need for a separate lexer can often be avoided (even for substantial grammars such as that for Gofer), with lexical issues being handled within the main parser by using some special purpose combinators.

6.1 White-space, comments, and keywords

We begin by defining a parser that consumes white-space from the beginning of a string, with a dummy value `()` returned as result:

```
spaces :: Parser ()
spaces  = [() | _ <- many1 (sat isSpace)]
          where
             isSpace x =
                (x == ' ') || (x == '\n') || (x == '\t')
```

Similarly, a single-line Gofer comment can be consumed as follows:

```
comment :: Parser ()
comment  = [() | _ <- string "--"
               , _ <- many (sat (\x -> x /= '\n'))]
```

We leave it as an exercise for the reader to define a parser for consuming multi-line Gofer comments {- ... -}, which can be nested.

After consuming white-space, there may still be a comment left to consume from the input string. Dually, after a comment there may still be white-space. Thus we are motivated to defined a special parser that repeatedly consumes white-space and comments until no more remain:

```
junk :: Parser ()
junk  = [() | _ <- many (spaces +++ comment)]
```

Note that while `spaces` and `comment` can fail, the `junk` parser always succeeds. We define two combinators in terms of `junk`: `parse` removes junk *before* applying a given parser, and `token` removes junk *after* applying a parser:

```
parse  :: Parser a -> Parser a
parse p = [v | _ <- junk, v <- p]

token  :: Parser a -> Parser a
token p = [v | v <- p, _ <- junk]
```

With the aid of these two combinators, parsers can be modified to ignore white-space and comments. Firstly, `parse` is applied once to the parser as a whole, ensuring that input to the parser begins at a significant character. And secondly, `token` is applied once to all sub-parsers that consume complete tokens, thus ensuring that the input always remains at a significant character.

Examples of parsers for complete tokens are `nat` and `int` (for natural numbers and integers), parsers of the form `string xs` (for symbols and keywords), and `ident` (for identifiers). It is useful to define special versions of these parsers — and more generally, special versions of any user-defined parsers for complete tokens — that encapsulate the necessary application of `token`:

```
natural      :: Parser Int
natural       = token nat

integer      :: Parser Int
integer       = token int

symbol       :: String -> Parser String
symbol xs     = token (string xs)

identifier   :: [String] -> Parser String
identifier ks = token [x | x <- ident, not (elem x ks)]
```

Note that `identifier` takes a list of keywords as an argument, where a keyword is a string that is not permitted as an identifier. For example, in Gofer the strings "`data`" and "`where`" (among others) are keywords. Without the keyword check, parsers defined in terms of `identifier` could produce unexpected results, or involve unnecessary backtracking to construct the correct parse of the input string.

6.2 A parser for λ-expressions

To illustrate the use of the new combinators given above, let us define a parser for simple λ-expressions extended with a "`let`" construct for local definitions. Parsed expressions will be represented in Gofer as follows:

```
data Expr = App Expr Expr            -- application
          | Lam String Expr          -- lambda abstraction
          | Let String Expr Expr     -- local definition
          | Var String               -- variable
```

Now a parser `expr :: Parser Expr` can be defined by:

```
expr     = atom `chainl1` [App]

atom     = lam +++ local +++ var +++ paren

lam      = [Lam x e | _ <- symbol "\\"
                    , x <- variable
                    , _ <- symbol "->"
                    , e <- expr]

local    = [Let x e e' | _  <- symbol "let"
                       , x  <- variable
                       , _  <- symbol "="
                       , e  <- expr
                       , _  <- symbol "in"
                       , e' <- expr]

var      = [Var x | x <- variable]

paren    = bracket (symbol "(") expr (symbol ")")

variable = identifier ["let","in"]
```

Note how the `expr` parser handles white-space and comments by using the `symbol` parser in place of `string` and `char`. Similarly, the keywords "`let`" and "`in`" are handled by using `identifier` to define the parser for variables. Finally, note how applications `(f e1 e2 ... en)` are parsed in the form `(((f e1) e2) ... )` by using the `chainl1` combinator.

7 Factorising the parser monad

Up to this point in the article, combinator parsers have been our only example of the notion of a monad. In this section we define a number of other monads related to the parser monad, leading up to a modular reformulation of the parser monad in terms of two simpler monads (Jones, 1995a). The immediate benefit is that, as

we shall see, the basic parser combinators no longer need to be defined explicitly. Rather, they arise automatically as a special case of lifting monad operations from a base monad m to a certain other monad parameterised over m. This also means that, if we change the nature of parsers by modifying the base monad (for example, limiting parsers to producing at most one result), new combinators for the modified monad of parsers are also defined automatically.

7.1 The exception monad

Before starting to define other monads, it is useful to first focus briefly on the intuition behind the use of monads in functional programming (Wadler, 1992a).

The basic idea behind monads is to distinguish the *values* that a computation can produce from the *computation* itself. More specifically, given a monad `m` and a type `a`, we can think of `m a` as the type of computations that yield results of type `a`, with the nature of the computation captured by the type constructor `m`. The combinators `result` and `bind` (with `zero` and `(++)` if appropriate) provide a means to structure the building of such computations:

```
result :: m a
bind   :: m a -> (a -> m b) -> m b
zero   :: m a
(++)   :: m a -> m a -> m a
```

From a computational point of view, `result` converts values into computations that yield those values; `bind` chains two computations together in sequence, with results of the first computation being made available for use in the second; `zero` is the trivial computation that does nothing; and finally, `(++)` is some kind of choice operation for computations.

Consider, for example, the type constructor `Maybe`:

```
data Maybe a = Just a | Nothing
```

We can think of a value of type `Maybe a` as a computation that either succeeds with a value of type `a`, or fails, producing no value. Thus, the type constructor `Maybe` captures computations that have the possibility to fail.

Defining the monad combinators for a given type constructor is usually just a matter of making the "obvious definitions" suggested by the types of the combinators. For example, the type constructor `Maybe` can be made into a monad with a zero and plus using the following definitions:

```
instance Monad Maybe where
   -- result        :: a -> Maybe a
   result x          = Just x

   -- bind          :: Maybe a -> (a -> Maybe b) -> Maybe b
   (Just x) `bind` f = f x
   Nothing  `bind` f = Nothing
```

```
instance MonadOPlus Maybe where
   -- zero          :: Maybe a
   zero              = Nothing

   -- (++)          :: Maybe a -> Maybe a -> Maybe a
   Just x  ++ y      = Just x
   Nothing ++ y      = y
```

That is, `result` converts a value into a computation that succeeds with this value; `bind` is a sequencing operator, with a successful result from the first computation being available for use in the second computation; `zero` is the computation that fails; and finally, `(++)` is a (deterministic) choice operator that returns the first computation if it succeeds, and the second otherwise.

Since failure can be viewed as a simple kind of exception, `Maybe` is sometimes called the *exception* monad in the literature (Spivey, 1990).

7.2 The non-determinism monad

A natural generalisation of `Maybe` is the list type constructor `[]`. While a value of type `Maybe a` can be thought of as a computation that either succeeds with a single result of type a or fails, a value of type `[a]` can be thought of as a computation that has the possibility to succeed with any number of results of type `a`, including zero (which represents failure). Thus the list type constructor `[]` can be used to capture *non-deterministic* computations.

Now `[]` can be made into a monad with a zero and plus:

```
instance Monad [] where
   -- result      :: a -> [a]
   result x        = [x]

   -- bind        :: [a] -> (a -> [b]) -> [b]
   []     `bind` f = []
   (x:xs) `bind` f = f x ++ (xs `bind` f)

instance MonadOPlus [] where
   -- zero        :: [a]
   zero            = []

   -- (++)        :: [a] -> [a] -> [a]
   []     ++ ys    = ys
   (x:xs) ++ ys    = x : (xs ++ ys)
```

That is, `result` converts a value into a computation that succeeds with this single value; `bind` is a sequencing operator for non-deterministic computations; `zero` always fails; and finally, `(++)` is a (non-deterministic) choice operator that appends the results of the two argument computations.

7.3 The state-transformer monad

Consider the (binary) type constructor `State`:

```
type State s a = s -> (a,s)
```

Values of type `State s a` can be interpreted as follows: they are computations that take an initial state of type `s`, and yield a value of type `a` together with a new state of type `s`. Thus, the type constructor `State s` obtained by applying `State` to a single type `s` captures computations that involve state of type `s`. We will refer to values of type `State s a` as *stateful computations*.

Now `State s` can be made into a monad:

```
instance Monad (State s) where
   -- result  :: a -> State s a
   result v    = \s -> (v,s)

   -- bind    :: State s a -> (a -> State s b) -> State s b
   st `bind` f = \s -> let (v,s') = st s in f v s'
```

That is, `result` converts a value into a stateful computation that returns that value without modifying the internal state, and `bind` composes two stateful computations in sequence, with the result value from the first being supplied as input to the second. Thinking pictorially in terms of boxes and wires is a useful aid to becoming familiar with these two operations (Jones & Launchbury, 1994).

The *state-transformer* monad `State s` does not have a zero and a plus. However, as we shall see in the next section, the *parameterised* state-transformer monad over a given based monad `m` *does* have a zero and a plus, provided that `m` does.

To allow us to access and modify the internal state, a few extra operations on the monad `State s` are introduced. The first operation, `update`, modifies the state by applying a given function, and returns the old state as the result value of the computation. The remaining two operations are defined in terms of `update`: `set` replaces the state with a new state, and returns the old state as the result; `fetch` returns the state without modifying it.

```
update  :: (s -> s) -> State s s
set     :: s -> State s s
fetch   :: State s s

update f = \s -> (s, f s)
set s    = update (\_ -> s)
fetch    = update id
```

In fact `State s` is not the only monad for which it makes sense to define these operations. For this reason we encapsulate the extra operations in a class, so that the same names can be used for the operations of different monads:

```
class Monad m => StateMonad m s where
   update :: (s -> s) -> m s
```

```
set    :: s -> m s
fetch  :: m s

set s   = update (\_ -> s)
fetch   = update id
```

This declaration can be read as follows: a type constructor `m` and a type `s` are together a member of the class `StateMonad` if `m` is a member of the class `Monad`, and if `m` is also equipped with `update`, `set`, and `fetch` operations of the specified types. Moreover, the fact that `set` and `fetch` can be defined in terms of `update` is also reflected in the declaration, by means of default definitions.

Now because `State s` is already a monad, it can be made into a state monad using the `update` operation as defined earlier:

```
instance StateMonad (State s) s where
   -- update :: (s -> s) -> State s s
   update f   = \s -> (s, f s)
```

7.4 The parameterised state-transformer monad

Recall now our type of combinator parsers:

```
type Parser a = String -> [(a,String)]
```

We see now that parsers combine two kinds of computation: non-deterministic computations (the result of a parser is a list), and stateful computations (the state is the string being parsed). Abstracting from the specific case of returning a list of results, the `Parser` type gives rise to a generalised version of the `State` type constructor that applies a given type constructor `m` to the result of the computation:

```
type StateM m s a = s -> m (a,s)
```

Now `StateM m s` can be made into a monad with a zero and a plus, by inheriting the monad operations from the base monad `m`:

```
instance Monad m => Monad (StateM m s) where
   -- result   :: a -> StateM m s a
   result v     = \s -> result (v,s)

   -- bind     :: StateM m s a ->
   --                (a -> StateM m s b) -> StateM m s b
   stm `bind` f = \s -> stm s `bind` \(v,s') -> f v s'

instance MonadOPlus m => MonadOPlus (StateM m s) where
   -- zero     :: StateM m s a
   zero         = \s -> zero

   -- (++)     :: StateM m s a -> StateM m s a -> StateM m s a
   stm ++ stm'  = \s -> stm s ++ stm' s
```

That is, `result` converts a value into a computation that returns this value without modifying the internal state; `bind` chains two computations together; `zero` is the computation that fails regardless of the input state; and finally, `(++)` is a choice operation that passes the same input state through to both of the argument computations, and combines their results.

In the previous section we defined the extra operations `update`, `set` and `fetch` for the monad `State s`. Of course, these operations can also be defined for the *parameterised state-transformer monad* `StateM m s`. As previously, we only need to define `update`, the remaining two operations being defined automatically via default definitions:

```
instance Monad m => StateMonad (StateM m s) s where
   -- update :: Monad m => (s -> s) -> StateM m s s
   update f   = \s -> result (s, f s)
```

7.5 The parser monad revisited

Recall once again our type of combinator parsers:

```
type Parser a = String -> [(a,String)]
```

This type can now be re-expressed using the parameterised state-transformer monad `StateM m s` by taking `[]` for `m`, and `String` for `s`:

```
type Parser a = StateM [] String a
```

But why view the `Parser` type in this way? The answer is that all the basic parser combinators no longer need to be defined explicitly (except one, the parser `item` for single characters), but rather arise as an instance of the general case of extending monad operations from a type constructor `m` to the type constructor `StateM m s`. More specifically, since `[]` forms a monad with a zero and a plus, so does `State [] String`, and hence Gofer automatically provides the following combinators:

```
result :: a -> Parser a
bind   :: Parser a -> (a -> Parser b) -> Parser b
zero   :: Parser a
(++)   :: Parser a -> Parser a -> Parser a
```

Moreover, defining the parser monad in this modular way in terms of `StateM` means that, if we change the type of parsers, then new combinators for the modified type are also defined automatically. For example, consider replacing

```
type Parser a = StateM [] String a
```

by a new definition in which the list type constructor `[]` (which captures non-deterministic computations that can return many results) is replaced by the `Maybe` type constructor (which captures deterministic computations that either fail, returning no result, or succeed with a single result):

```
data Maybe a  = Just a | Nothing

type Parser a = StateM Maybe String a
```

Since `Maybe` forms a monad with a zero and a plus, so does the re-defined `Parser` type constructor, and hence Gofer automatically provides `result`, `bind`, `zero`, and `(++)` combinators for deterministic parsers. In earlier approaches that do not exploit the monadic nature of parsers (Wadler, 1985; Hutton, 1992; Fokker, 1995), the basic combinators would have to be re-defined by hand.

The only basic parsing primitive that does not arise from the monadic structure of the `Parser` type is the parser `item` for consuming single characters:

```
item :: Parser Char
item  = \inp -> case inp of
                  []     -> []
                  (x:xs) -> [(x,xs)]
```

However, `item` can now be re-defined in monadic style. We first fetch the current state (the input string); if the string is empty then the `item` parser fails, otherwise the first character is consumed (by applying the `tail` function to the state), and returned as the result value of the parser:

```
item  = [x | (x:_) <- update tail]
```

The advantage of the monadic definition of `item` is that it does not depend upon the internal details of the `Parser` type. Thus, for example, it works equally well for both the non-deterministic and deterministic versions of `Parser`.

8 Handling the offside rule

Earlier (section 6) we showed that the need for a lexer to handle white-space, comments, and keywords can be avoided by using special combinators within the main parser. Another task usually performed by a lexer is handling the Gofer *offside rule*. This rule allows the grouping of definitions in a program to be indicated using indentation, and is usually implemented by the lexer inserting extra tokens (concerning indentation) into its output stream.

In this section we show that Gofer's offside rule can be handled in a simple and natural manner without a separate lexer, by once again using special combinators. Our approach was inspired by the monadic view of parsers, and is a development of an idea described earlier by Hutton (1992).

8.1 The offside rule

Consider the following simple Gofer program:

```
a = b + c
    where
       b = 10
```

```
       c = 15 - 5
d = a * 2
```

It is clear from the use of indentation that `a` and `d` are intended to be global definitions, with `b` and `c` local definitions to `a`. Indeed, the above program can be viewed as a shorthand for the following program, in which the grouping of definitions is made explicit using special brackets and separators:

```
{ a = b + c
      where
         { b = 10
         ; c = 15 - 5 }
; d = a * 2 }
```

How the grouping of Gofer definitions follows from their indentation is formally specified by the *offside rule*. The essence of the rule is as follows: consecutive definitions that begin in the same column c are deemed to be part of the same group. To make parsing easier, it is further required that the remainder of the text of each definition (excluding white-space and comments, of course) in a group must occur in a column strictly greater than c. In terms of the offside rule then, definitions `a` and `d` in the example program above are formally grouped together (and similarly for `b` and `c`) because they start in the same column as one another.

8.2 Modifying the type of parsers

To implement the offside rule, we will have to maintain some extra information during parsing. First of all, since column numbers play a crucial role in the offside rule, parsers will need to know the column number of the first character in their input string. In fact, it turns out that parsers will also require the current line number. Thus our present type of combinator parsers,

```
type Parser a = StateM [] String a
```

is revised to the following type, in which the internal state of a parser now contains a (line,column) position in addition to a string:

```
type Parser a = StateM [] Pstring a

type Pstring  = (Pos,String)

type Pos      = (Int,Int)
```

In addition, parsers will need to know the starting position of the current definition being parsed — if the offside rule is not in effect, this *definition position* can be set with a negative column number. Thus our type of parsers is revised once more, to take the current definition position as an extra argument:

```
type Parser a = Pos -> StateM [] Pstring a
```

Another option would have been to maintain the definition position in the parser state, along with the current position and the string to be parsed. However, definition positions can be nested, and supplying the position as an extra argument to parsers — as opposed to within the parser state — is more natural from the point of view of implementing nesting of positions.

Is the revised `Parser` type still a monad? Abstracting from the details, the body of the `Parser` type definition is of the form `s -> m a` (in our case `s` is `Pos`, `m` is the monad `StateM [] Pstring`, and `a` is the parameter type `a`.) We recognise this as being similar to the type `s -> m (a,s)` of parameterised state-transformers, the difference being that the type `s` of states no longer occurs in the type of the result: in other words, the state can be read, but not modified. Thus we can think of `s -> m a` as the type of *parameterised state-readers*. The monadic nature of this type is the topic of the next section.

8.3 The parameterised state-reader monad

Consider the type constructor `ReaderM`, defined as follows:

```
type ReaderM m s a = s -> m a
```

In a similar way to `StateM m s`, `ReaderM m s` can be made into a monad with a zero and a plus, by inheriting the monad operations from the base monad `m`:

```
instance Monad m => Monad (ReaderM m s) where
   -- result   :: a -> ReaderM m s a
   result v     = \s -> result v

   -- bind     :: ReaderM m s a ->
   --                (a -> ReaderM m s b) -> ReaderM m s b
   srm `bind` f = \s -> srm s `bind` \v -> f v s

instance MonadOPlus m => MonadOPlus (ReaderM m s) where
   -- zero     :: ReaderM m s a
   zero         = \s -> zero

   -- (++)     :: ReaderM m s a ->
   --                ReaderM m s a -> ReaderM m s a
   srm ++ srm'  = \s -> srm s ++ srm' s
```

That is, `result` converts a value into a computation that returns this value without consulting the state; `bind` chains two computations together, with the same state being passed to both computations (contrast with the `bind` operation for `StateM`, in which the second computation receives the new state produced by the first computation); `zero` is the computation that fails; and finally, `(++)` is a choice operation that passes the same state to both of the argument computations.

To allow us to access and set the state, a couple of extra operations on the *parameterised state-reader* monad `ReaderM m s` are introduced. As for `StateM`, we

encapsulate the extra operations in a class. The operation `env` returns the state as the result of the computation, while `setenv` replaces the current state for a given computation with a new state:

```
class Monad m => ReaderMonad m s where
   env    :: m s
   setenv :: s -> m a -> m a

instance Monad m => ReaderMonad (ReaderM m s) s where
   -- env      :: Monad m => ReaderM m s s
   env          = \s -> result s

   -- setenv   :: Monad m => s ->
   --                ReaderM m s a -> ReaderM m s a
   setenv s srm = \_ -> srm s
```

The name `env` comes from the fact that one can think of the state supplied to a state-reader as being a kind of *env*ironment. Indeed, in the literature state-reader monads are sometimes called *environment* monads.

8.4 The new parser combinators

Using the `ReaderM` type constructor, our revised type of parsers

```
type Parser a = Pos -> StateM [] Pstring a
```

can now be expressed as follows:

```
type Parser a = ReaderM (StateM [] Pstring) Pos a
```

Now since `[]` forms a monad with a zero and a plus, so does `StateM [] Pstring`, and hence so does `ReaderM (StateM [] Pstring) Pos`. Thus Gofer automatically provides `result`, `bind`, `zero`, and `(++)` operations for parsers that can handle the offside rule. Since the type of parsers is now defined in terms of `ReaderM` at the top level, the extra operations `env` and `setenv` are also provided for parsers. Moreover, the extra operation `update` (and the derived operations `set` and `fetch`) from the underlying state monad can be lifted to the new type of parsers — or more generally, to any parameterised state-reader monad — by ignoring the environment:

```
instance StateMonad m a => StateMonad (ReaderM m s) a where
   -- update :: StateMonad m a => (a -> a) -> ReaderM m s a
   update f   = \_ -> update f
```

Now that the internal state of parsers has been modified (from `String` to `Pstring`), the parser `item` for consuming single characters from the input must also be modified. The new definition for `item` is similar to the old,

```
item :: Parser Char
item  = [x | (x:_) <- update tail]
```

except that the `item` parser now fails if the position of the character to be consumed is not *onside* with respect to current definition position:

```
item :: Parser Char
item  = [x | (pos,x:_) <- update newstate
           , defpos    <- env
           , onside pos defpos]
```

A position is onside if its column number is strictly greater than the current definition column. However, the first character of a new definition begins in the same column as the definition column, so this is handled as a special case:

```
onside              :: Pos -> Pos -> Bool
onside (l,c) (dl,dc) = (c > dc) || (l == dl)
```

The remaining auxiliary function, `newstate`, consumes the first character from the input string, and updates the current position accordingly (for example, if a newline character was consumed, the current line number is incremented, and the current column number is set back to zero):

```
newstate :: Pstring -> Pstring
newstate ((l,c),x:xs)
   = (newpos,xs)
     where
        newpos = case x of
                    '\n' -> (l+1,0)
                    '\t' -> (l,((c `div` 8)+1)*8)
                    _    -> (l,c+1)
```

One aspect of the offside rule still remains to be addressed: for the purposes of this rule, white-space and comments are not significant, and should always be successfully consumed even if they contain characters that are not onside. This can be handled by temporarily setting the definition position to $(0, -1)$ within the `junk` parser for white-space and comments:

```
junk :: Parser ()
junk  = [() | _ <- setenv (0,-1) (many (spaces +++ comment))]
```

All that remains now is to define a combinator that parses a sequence of definitions subject to the Gofer offside rule:

```
many1_offside  :: Parser a -> Parser [a]
many1_offside p = [vs | (pos,_) <- fetch
                      , vs      <- setenv pos (many1 (off p))]
```

That is, `many1_offside p` behaves just as `many1 (off p)`, except that within this parser the definition position is set to the current position. (There is no need to skip white-space and comments before setting the position, since this will already have been effected by proper use of the lexical combinators `token` and `parse`.) The auxiliary combinator `off` takes care of setting the definition position locally for

each new definition in the sequence, where a new definition begins if the column position equals the definition column position:

```
off  :: Parser a -> Parser a
off p = [v | (dl,dc)   <- env
           , ((l,c),_) <- fetch
           , c == dc
           , v         <- setenv (l,dc) p]
```

For completeness, we also define a combinator `many_offside` that has the same behaviour as the combinator `many1_offside`, except that it can also parse an empty sequence of definitions:

```
many_offside :: Parser a -> Parser [a]
many_offside p = many1_offside p +++ [[]]
```

To illustrate the use of the new combinators defined above, let us modify our parser for λ-expressions (section 6.2) so that the "`let`" construct permits non-empty sequences of local definitions subject to the offside rule. The datatype `Expr` of expressions is first modified so that the `Let` constructor has type `[(String,Expr)] -> Expr` instead of `String -> Expr -> Expr`:

```
data Expr = ...
          | Let [(String,Expr)] Expr
          | ...
```

The only part of the parser that needs to be modified is the parser `local` for local definitions, which now accepts sequences:

```
local = [Let ds e | _  <- symbol "let"
                  , ds <- many1_offside defn
                  , _  <- symbol "in"
                  , e  <- expr]

defn  = [(x,e) | x <- identifier
               , _ <- symbol "="
               , e <- expr]
```

We conclude this section by noting that the use of the offside rule when laying out sequences of Gofer definitions is not mandatory. As shown in our initial example, one also has the option to include explicit layout information in the form of parentheses "{" and "}" around the sequence, with definitions separated by semi-colons ";". We leave it as an exercise to the reader to use `many_offside` to define a combinator that implements this convention.

In summary then, to permit combinator parsers to handle the Gofer offside rule, we changed the type of parsers to include some positional information, modified the `item` and `junk` combinators accordingly, and defined two new combinators: `many1_offside` and `many_offside`. All other necessary redefining of combinators is done automatically by the Gofer type system.

9 Acknowledgements

The first author was employed by the University of Utrecht during part of the writing of this article, for which funding is gratefully acknowledged.

Special thanks are due to Luc Duponcheel for many improvements to the implementation of the combinator libraries in Gofer (particularly concerning the use of type classes and restricted type synonyms), and to Mark P. Jones for detailed comments on the final draft of this article.

10 Appendix: a parser for data definitions

To illustrate the monadic parser combinators developed in this article in a real-life setting, we consider the problem of parsing a sequence of Gofer datatype definitions. An example of such a sequence is as follows:

```
data List a = Nil | Cons a (List a)

data Tree a b = Leaf a
              | Node (Tree a b, b, Tree a b)
```

Within the parser, datatypes will be represented as follows:

```
type Data = (String,             -- type name
             [String],           -- parameters
             [(String,[Type])])  -- constructors and arguments
```

The representation `Type` for types will be treated shortly. A parser `datadecls :: Parser [Data]` for a sequence of datatypes can now be defined by

```
datadecls   = many_offside datadecl

datadecl    = [(x,xs,b) | _  <- symbol "data"
                        , x  <- constructor
                        , xs <- many variable
                        , _  <- symbol "="
                        , b  <- condecl `sepby1` symbol "|"]

constructor = token [(x:xs) | x  <- upper
                            , xs <- many alphanum]

variable    = identifier ["data"]

condecl     = [(x,ts) | x  <- constructor
                      , ts <- many type2]
```

There are a couple of points worth noting about this parser. Firstly, all lexical issues (white-space and comments, the offside rule, and keywords) are handled by combinators. And secondly, since `constructor` is a parser for a complete token, the `token` combinator is applied within its definition.

Within the parser, types will be represented as follows:

```
data Type   = Arrow Type Type   -- function
            | Apply Type Type   -- application
            | Var String        -- variable
            | Con String        -- constructor
            | Tuple [Type]      -- tuple
            | List Type         -- list
```

A parser `type0 :: Parser Type` for types can now be defined by

```
type0       = type1 `chainr1` [Arrow | _ <- symbol "->"]
type1       = type2 `chainl1` [Apply]
type2       = var +++ con +++ list +++ tuple

var         = [Var x | x <- variable]

con         = [Con x | x <- constructor]

list        = [List x | x <- bracket
                              (symbol "[")
                              type0
                              (symbol "]")]

tuple       = [f ts | ts <- bracket
                              (symbol "(")
                              (type0 `sepby` symbol ",")
                              (symbol ")")]
              where f [t] = t
                    f ts  = Tuple ts
```

Note how `chainr1` and `chainl1` are used to handle parsing of function-types and application. Note also that (as in Gofer) building a singleton tuple `(t)` of a type `t` is not possible, since `(t)` is treated as a parenthesised expression.

References

Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers — principles, techniques and tools*. Addison-Wesley.

Burge, W.H. (1975). *Recursive programming techniques*. Addison-Wesley.

Fokker, Jeroen. 1995 (May). Functional parsers. *Lecture notes of the Baastad Spring school on functional programming*.

Gill, Andy, & Marlow, Simon. 1995 (Jan.). *Happy: the parser generator for Haskell*. University of Glasgow.

Hughes, John. (1989). Why functional programming matters. *The computer journal*, **32**(2), 98–107.

Hutton, Graham. (1992). Higher-order functions for parsing. *Journal of functional programming*, **2**(3), 323–343.

Jones, Mark P. (1994). *Gofer 2.30a release notes*. Unpublished manuscript.

Jones, Mark P. (1995a). Functional programming beyond the Hindley/Milner type system. *Proc. lecture notes of the Baastad spring school on functional programming*.

Jones, Mark P. (1995b). *The Gofer distribution*. Available from the University of Nottingham: `http://www.cs.nott.ac.uk/Department/Staff/mpj/`.

Jones, Mark P. (1995c). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.

Jones, Simon Peyton, & Launchbury, John. (1994). *State in Haskell*. University of Glasgow.

Landin, Peter. (1966). The next 700 programming languages. *Communications of the ACM*, **9**(3).

Mogensen, Torben. (1993). *Ratatosk: a parser generator and scanner generator for Gofer*. University of Copenhagen (DIKU).

Moggi, Eugenio. (1989). Computation lambda-calculus and monads. *Proc. IEEE symposium on logic in computer science*. A extended version of the paper is available as a technical report from the University of Edinburgh.

Röjemo, Niklas. (1995). *Garbage collection and memory efficiency in lazy functional languages*. Ph.D. thesis, Chalmers University of Technology.

Spivey, Mike. (1990). A functional theory of exceptions. *Science of computer programming*, **14**, 25–42.

Wadler, Philip. (1985). How to replace failure by a list of successes. *Proc. conference on functional programming and computer architecture*. Springer–Verlag.

Wadler, Philip. (1990). Comprehending monads. *Proc. ACM conference on Lisp and functional programming*.

Wadler, Philip. (1992a). The essence of functional programming. *Proc. principles of programming languages*.

Wadler, Philip. (1992b). Monads for functional programming. Broy, Manfred (ed), *Proc. Marktoberdorf Summer school on program design calculi*. Springer–Verlag.

Why Functional Programming Matters

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

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!*

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