CPSC 449:
Notes on hand evaluating Haskell programs
(for discussion in labs!)

Robin Cockett

September 13, 2022

1 Hand evaluation

Why hand evaluate a Haskell program when you have a machine which can do it for you? Well the purpose is that you should understand what happens when you run a program and this is really useful when you expect one behaviour and get another behaviour. Knowing how to hand trace a program allows you to understand what has led to that unexpected behaviour. Haskell, of course, is quite exceptional as it has a very predictable behaviour which is supported by its mathematical underpinnings. Nonetheless, there are some important aspects of its "lazy evaluation" which one can see from hand evaluations and may already be not quite what you may have expect.

Of course, you will not be able to hand evaluate a large Haskell program! However, if it is written in a modular fashion you should be able to hand simulate (if only in your mind) key routines where one assumes that the calls to other components return the correct answer. Thus, for debugging, hand evaluation is a very basic and important tool onto which one can fall back.

2 Fibonacci fun

Let us consider as a starting example the Fibonacci function written in its most basic form

```
fib:: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = (fib (n-1)) + (fib (n-2))
```

Suppose I want to calculate `fib 3`. Here is a hand evaluation:

```
|fib 3?
|fib 3 =: fib 0
           ...match fails
|fib 3 =: fib 1
           ...match fails
```

```
|fib 3 =: fib n
           ...match [n := 3 ]
| =(fib (n-1)) + (fib (n-2)) [n:=3]          % substituted RHS
| =(fib (3-1)) + (fib (3-2)) = (fib 2) + (fib 1)
|  |fib 2?  % evaluate 1st arg
|  |fib 2 =: fib 0
           ...match fails
|  |fib 2 =: fib 1
           ...match fails
|  |fib 2 =: fib n
           ...match [n:=2]
|  | fib (n-1)) + (fib (n-2)) [n:=2]     % substituted RHS
|  | =(fib (2-1)) + (fib (2-2))
|  | =(fib 1) + (fib 0)
|  |  |fib   1?
|  |  |fib 1 =: fib 0
           ...match fails
|  |  |fib 1 =: fib 1
           ...match []
|  |  |1        % call substituted RHS (return)
|  |1+ (fib 0)
|  |  |fib 0?
|  |  |fib 0 =: fib 0
           ...match []
|  |  |0     % call substituted RHS (return)
|  |1+0 = 1
|1 + fib 1
|  |fib 1?  % evaluate second argument
|  |     ...
|  |1 % return
|1+1?
|2
```

Notice even though this is a relatively small calculation I have skipped over some steps (the ellipsis) because there is, even for this program, quite a lot happening! Let us discuss what happens:

We start by asking what `fib 3` evaluates to. To determine this one must work through the phrases or patterns of the fibonacci function trying to match each in turn to the input. Is the value at which one is evaluating 0? Is it 1? Both these matching attempts fail and one drops through to the third case. Here `fib 3` is matched to the pattern `fib n` (we write this `fib 3 =:  fib n`): the match succeeds producing the substitution $n := 3$ of the pattern: the righthand side of the third phrase is then called with the substitution $n := 3$.

This leaves the evaluation of `(fib 2) + (fib 1)`. Notice that I have "eagerly" evaluated the arithmetic expressions to keep things simple - in fact even these expressions may be evaluated "lazily". The next aim is then to evaluate the left argument of the addition: so we need to evaluate `fib 2`. To evaluate this one needs to go through the phrases of the function trying, in turn, to

match each phrase of the function. Does `0` match `2`? As this is false so one proceeds to the next pattern and `1` fails to match `2`. So one tries to match the last pattern which succeeds and produces the substitution `n:= 2`. One then recursively calls the substituted righthand side as `(fib 1) + (fib 0)`. One first evaluates the lefthand argument of this addition (this time the second pattern succeeds and returns `1`) then the right hand argument (which returns `0`) these are summed and passed up.

One might ask whether this hand evaluation is *exactly* what Haskell does ... and the answer is not quite. This is intended as a "conceptual" view of evaluation: Haskell "compiles" this conceptual view to make it much more efficient. Despite this the hand evaluations reflect quite well what actually happens hence its value.

The order in which the evaluation proceeds is important and determines the behaviour. Patterns which are basic values (such as integers) are resolved by equality tests. Other patterns are resolved by *matching* (hence the name "pattern matching"): we will discuss this more below. Let us first consider consider the (improved) Fibonacci program which checks that the input is a positive number. This program uses *guards*: this is another powerful syntactic feature of Haskell and it is useful to understand how they are evaluated.

```
fib:: Integer -> Integer
fib n
   | n < 0     = error "Positive numbers only please ..."
   | n == 0    = 0
   | n == 1    = 1
   | otherwise = (fib (n-1)) + (fib (n-2))
```

Here is a trace of `fib 3`:

```
|fib 3
|fib 3 =: fib n
           ...match [n:= 3]
|  n < 0 [n:=3] = 3 < 0
             ...false
|  n == 0 [n:=3] = 3==0
             ...false
|  n == 1 [n:=3] = 3 == 1
             ...false
|  otherwise
             ...true
|  = fib(n-1) + fib(n-2) [n:=3] = fib 2 + fib 1   % call  rhs
|   |fib 2 =: fib n  %evaluate 1st argument
           ...match [n:= 2]
|   |  n < 0  [n:= 3-1] = 2 < 0
            ...false
|   |  n == 0 [n:= 2] = 2 == 0
            ...false
|   |  n == 1 [n: = 2] = 2 == 1
```

```
            ...false
|   |  otherwise
            ...true
|   |   fib(n-1) + fib(n-2)[n:=2] =  fib 1 + fib 0?  % call rhs
|   |   |fib 1 =: fib n % evaluate 1st argument
           ...match [n:= 1]
|   |   | 1 < 0 ...false
|   |   | 1 == 0 ...false
|   |   | 1 == 1 ...true
|   |   |1                     % evaluate RHS of guard
|   | 1 + fib 0
|   |   |fib 0 % evaluate  2nd argument
|   |   |  0 <  0?  ...false
|   |   |  0 == 0? ...true
|   |   |0                   % evaluate RHS of guard
|   |1 + 0?
|   |1
|1 + fib 1
|   |fib 1
|   |  1 < 0? ...false
|   |  1 == 0 ...false
|   |  1 == 1 ...true
|   |1                   % evaluate RHS of guard
|1 + 1?
|2
```

So, of course, this is very similar to the previous evaluation. However, notice how the guards are evaluated: each guard is evaluated sequentially until a guard which evaluates to true is found. The first true guard then has its body evaluated. The last guard, the `otherwise` is always true and so is the default action when all the other guards are false.

Note in this hand evaluation I have assumed that arithmetic expressions get evaluated immediately (strict evaluation) in order to simplify the evaluation. This is not actually what happens as even these expression are not evaluated until they are needed. Thus, their evaluation is initiated when the guards have to be evaluated.

We know this is a *very* inefficient way of evaluating the Fibonacci function. To see how bad this is we can count the recursive calls we have to `fib`:

$$\mathsf{calls}[\mathsf{fib}](n) = \mathsf{calls}[\mathsf{fib}](n-1) + \mathsf{calls}[\mathsf{fib}](n-2) + 1 \quad \text{when } n > 2$$

$$\mathsf{calls}[\mathsf{fib}](0) = 1 \quad \text{and} \quad \mathsf{calls}[\mathsf{fib}](1) = 1 \quad \text{and} \quad \mathsf{calls}[\mathsf{fib}](2) = 2$$

By rewriting the code to terminate at $\mathsf{fib}(1)$ and $\mathsf{fib}(2)$:

```
fib 1 = 1
fib 2 = 1
fib n = (fib (n-1)) + (fib (n-2))
```

we can observe that the number of calls to fib is determined by:

$$\mathsf{calls}[\mathsf{fib}](n) = \mathsf{calls}[\mathsf{fib}](n-1) + \mathsf{calls}[\mathsf{fib}](n-2) + 1 \quad \text{when } n > 2$$

$$\mathsf{calls}[\mathsf{fib}](1) = 1 \quad \text{and} \quad \mathsf{calls}[\mathsf{fib}](2) = 1$$

where we now insist the input has $n > 0$. We claim that the number of calls to fib by fib n is:

$$\mathsf{calls}[\mathsf{fib}](n) = 2 \cdot (\mathsf{fib}(n)) - 1$$

We can prove this by induction:

Base: $\mathsf{calls}[\mathsf{fib}](1) = 2 \cdot \mathsf{fib}(1) - 1 = 2 - 1 = 1$ and $\mathsf{calls}[\mathsf{fib}](2) = 2 \cdot \mathsf{fib}(2) - 1 = 2 - 1 = 1$.

Inductive step:

$$\begin{aligned}
\mathsf{calls}[\mathsf{fib}](n+2) &= \mathsf{calls}[\mathsf{fib}](n+1) + \mathsf{calls}[\mathsf{fib}](n) + 1 \\
&= 2 \cdot \mathsf{fib}(n+1) - 1 + 2 \cdot \mathsf{fib}(n) - 1 + 1 \\
&= 2 \cdot (\mathsf{fib}(n+1) + \mathsf{fib}(n)) - 1 \\
&= 2 \cdot (\mathsf{fib}(n+2) - 1
\end{aligned}$$

However, one can show $\mathsf{fib}(n) \geq G^n/\sqrt{5} - 1$, where G is the "golden ratio", $G = (1+\sqrt{5})/2$ (see notes on structural induction), this shows that the complexity of fib is exponential .

A *much* more efficient form of the Fibonacci function, which runs in linear time, is:

```
fib:: Integer -> Integer
fib n
  | n < 0 = error "Fibonacci only takes positive integers"
  | otherwise = pfib n (0,1)
 where
  pfib 0 (n,m) = n
  pfib r (n,m) = pfib (r-1) (fibstep (n,m))

  fibstep (n,m) = (m,n+m)
```

Notice how fibstep uses the type of pairs: it has type $(\mathsf{Integer}, \mathsf{Integer}) \to (\mathsf{Integer}, \mathsf{Integer})$. Let us trace this program for fib 3:

```
|fib 3?
| 3 < 0 ...false
| otherwise  ...true
|  =pfib 3 (0,1)?  call
|  |pfib 3 (0,1) =: pfib 0 (n,m)
|            ...match  fails  (0 :=\= 3)
|  |pfib 3 (0,1) =: pfib r (n,m)
|            ...match [r:=3,n:=0,m:=1]
|  |  |(pfib (r-1) (fibstep (n,m))[r:=3,n:=0,m:=1]?  %call pfib
```

```
|   |   |=pfib (3-1) (fibstep (0,1))
|   |   |  pfib 2 (fibstep (0,1)) =: pfib 0 (n,m)
              ...match fails 0=/=2
|   |   |  pfib 2 (fibstep (0,1))=:pfib r (n,m)
|   |   |  |fibstep (0,1)? %evaluate fibstep
|   |   |  | (m,n+m)[n:=0,m:=1] = (1,0+1) = (1,1)
|   |   |  |(1,1) return fibstep (0,1)
|   |   |pfib 2 (1,1) =:  pfib r (n,m)
              ...match [r:=2,n:=1,m:=1]
|   |   |  =pfib (r-1) (fibstep (n,m))[r:=2,n:=1,m:=1] ? % call pfib
|   |   |  =pfib (2-1) (fibstep (1,1))
|   |   |  |  pfib 1 (fibstep (1,1))  =: pfib 0 (n,m)
              ...fmatch fails 0:=/= 1
|   |   |  |  pfib 1 (fibstep (1,1))  =: pfib r (n,m)
|   |   |  |  |fibstep (1,1)=: fiibstep (n,m) %evaluate 2nd arg
              ...match [n:=1,m:= 1]?
|   |   |  |  | (m,n+m)[n:=1,m:=1] = (1,1+1) =(1,2)
|   |   |  |  |(1,2) % return fibstep (1,1)
|   |   |  |pfib 1 (1,2)  =: pfib r (n,m)
              ...match [r:=1,n:=1,m:=2]
|   |   |  |  =(pfib (r-1) (fibstep (n,m))) [r:=1,n:=1,m:=2]?
|   |   |  |  =pfib (1-1) (fibstep (1,2))
|   |   |  |  |pfib 0 (fibstep (1,2)) =: pfib 0 (n,m)
|   |   |  |  |  |fibstep (1,2) =: fibstep (n,m) match [n:=1,m:=2]?
|   |   |  |  |  |  (m,n+m)[n:=1,m:=2] = (2,1+2) = (2,3)
|   |   |  |  |  |(2,3) % return fibstep (1,2)
|   |   |  |  |pfib 0 (2,3) =: pfib 0 (n,m)
              ...fmatch [n:=2,n:=3]
|   |   |  |  |  n [n:=2,n:=3] = 2
|   |   |  |  |2  % return  (pfib 0 (2,3))
|   |   |  |2 % return  (pfib 1 (1,2))
|   |   |2 % return (pfib 2 (1,1))
|   |2 % return (pfib 3 (01))
|2 % return (fib 3)
```

From calculating a small Fibonacci number such as fib 3 one cannot really tell by comparing the traces that this is more efficient. However by looking at calls to pfib, it is immediately obvious that for fib n there are $n+1$ calls to pfib and, assuming constant time addition, each such call takes constant time. Thus, the complexity of this algorithm is linear.

Don't believe me? Try a comparison: run the two program on biggish numbers!

This is a sharp reminder that a "high level language", such as Haskell, does not allow one to escape the responsibility of writing good code and using efficient algorithms! What languages such as Haskell do allow is to capture algorithms very clearly and concisely. This, in turn, can allow one to understand the key aspects of algorithms better and helps to ensures they are implemented correctly. This clarity does come at cost: Haskell programs generally require more memory and

are not as efficient as their counterparts in an optimized imperative language (such as C). However, the difference is (roughly) a constant factor, thus considering that modern computers do not lack for memory or raw processing power this aspect is becoming increasingly less significant. The production of reasonably efficient programs which are correct while minimizing development costs remains, by way of contrast, a significant issue ... and is exactly the strength of high level languages such as Haskell.

3 Matching

In the evaluation of a Haskell program there are a number of steps which require *matching*. Matching is an algorithm which takes in two terms (ordered trees). One of these is the *pattern* (or a *template*) and can have variables at the leaves all of which must be distinct. The other is sometimes called the *subject* term: the purpose is to determine whether the subject term matches the pattern term. Explicitly the matching algorithm determines whether there is a substitution of the variables of the pattern term which will make the two terms equal. For example in

```
fib 42 =: fib n
```

`fib 42` is the subject term and `fib n` is the pattern term (which has the variable `n`). The matching algorithm in this case succeeds and produces the substitution `[n:=42]`. Matching can, of course fail. For example the matching algorithm for `fib 42 =: fib 0` fails: here the pattern term has no variable so the subject term must equal the pattern if the matching is to succeed.

The matching algorithm is quite simple and can be described recursively by:

(A) If the pattern is a variable return the substitution of that variable by the subject: $t =: x = [x := t]$.

(B) If the pattern starts with a term constructor (a function symbol) with a number of arguments the subject term must start in exactly the same way otherwise matching fails. The arguments are then recursively matched and the union of the substitutions generated for each argument returned. Thus $F(p_1, ..., p_n) := F(t_1, ..., t_n) = \bigcup_{i=1...n} p_i := t_i$.

Here are some examples:

(1) Consider first $F(t_1, t_2) =: F(x, x)$. This is an illegal matching problem as the pattern, $f(x, x)$ has a repeated variable (namely x).

(2) Consider $F(t_1, t_2) =: F(x, y)$: this matching succeeds and returns the substitution $[x := t_1, y := t_2]$.

(3) Consider $F(x, z) =: F(G(u, v), y)$: in this case we get the two matching subproblems, one for each argument, $x =: G(u, v)$ and $z := y$: the first of which fails as the variable x cannot be obtained by substituting the variables of $G(u, v)$.

(4) Consider $F(G(t_1, t_2), t_3) =: F(x, y)$: this match succeeds and returns the substitution $[x := G(t_1, t_2), y; = t_3]$.

(5) Consider $F(G(t_1, t_2), t_3) =: F(G(u, v), y)$: this match also succeeds and returns the substitution $[u := t_1, v := t_2, y; = t_3]$.

4 Evaluation order and Scott's bottom

Scott's bottom[1], often denoted $\perp$, is a special "element" of each type which never terminates. It is, thus, an "undefined" or "divergent" element. Any program which causes bottom to be evaluated will also not terminate. This means that one can test whether a program "touches" (or tries to evaluate) an argument using this function! This is rather useful in understanding how Haskell evaluates programs.

Here is the program for "bottom":

```
bottom::a
bottom = bottom
```

To see how we can use this to understand the behaviour of programs consider various definitions of "and":

```
data Bool = True | False
        deriving (Eq,Show)

-- The prelude's "and"

(&&):: Bool -> Bool -> Bool
(&&) True b  = b
(&&) _     _   = False

-- Three different versions of "and":

myand2:: Bool -> Bool -> Bool
myand2 True True = True
myand2    _     _     = False

myand2:: Bool -> Bool -> Bool
myand2 True True = True
myand2 True False = False
myand2 False True = False
myand2 False False = False

myand3:: Bool -> Bool -> Bool
myand3 _ False = False
myand3 False _ = False
myand3 True True = True
```

Now try running:

```
False && bottom
```

[1] It is named after the great computer scientist, philosopher, and mathematician Dana Scott.

Here is the hand evaluation:

```
| False && bottom
| False && bottom =: True && n
          ...match fails True =/=: False
|  _ && _ =: False && bottom
          ...match succeeds [] % no substitutions
|False    % evaluate RHS
```

Notice that the second argument is never evaluated. The matching process is sequential: first it tries to match the operation (&&) then the first arguments, then the second arguments etc. In this case it ties match on && and then tries `True=:False` this fails and the match against the first pattern phrase of the function fails. Next, after matching against &&, comes the matching of `False` against an "anonymous" variable: this always succeeds. Matching against an anonymous variable, or indeed any variable, as the pattern *always* succeeds, forces no further evaluation, and it generates no substitutions. Next is the matching of `bottom` against an "anonymous variable": this also succeeds and generates no substitutions. So one can evaluate the RHS of the second pattern phrase: which is `False`. The point is that at no stage is it necessary to evaluate `bottom`.

Consider now two different ways of calculating the conjunction of a list of Booleans. The first used a simple recursion

```
and:: [Bool] -> Bool
and [] = True
and (b:bs) = b && (and bs)
```

Let us hand evaluate `and [True,False,True,True]`:

```
|and [True,False,True,True] =: and []
             ...match fails
|and [True,False,True,True] =: and (b:bs)
             ...match [b:=True,bs:=[False,True,True]
|  =True & (and [False,True,True])
|  | True & (and [False,True,True])=:True & b
            ...match...  [b:= and [False,True,True]]
|  | b[[b:= and [False,True,True]]
|  |  =and [False,True,True]
|  |  | and [False,True,True] =: and []
             ...match fails
|  |  |and [False,True,True] =: and (b:bs)
               ...match [b:=False,bs:=[True,True]]
|  |  | =False & (and [True,True])
|  |  |  |False & (and [True,True])=:True & b
              ...match fails
|  |  |  |False & (and [True,True])=:_&_
```

```
                ...match []
 |  |  |  |False
 |  |  |False
 |  |False
 |False
```

Notice how the function never looks at the end of the list. Thus, `and [True,False,bottom]` also evaluates to `False` as the evaluation never touches the `bottom` function: I encourage you to check that this is the case by actually running the function in Haskell ...

Here is an alternative more sophisticated way of defining `and` using a fold:

```
foldr f g [] = g
foldr f g (a:as) = f a (fold f g as)

and = foldr (&&) True
```

Note that I have used the "section" `(&&)` which allows the non-infix form of `&&` which is a function `(&&):: Bool->Bool->Bool` of the required type for this fold.

Let us hand evaluate `and [True,False,True,True]` defined this way:

```
|and [True,False,True,True]
|=foldr (&&) True [True,False,True,True]
|  |foldr (&&) True [True,False,True,True]=:foldr f g []
                  ...match fails
|  |foldr (&&) True [True,False,True,True] =: foldr f g (a:as)
                  ...match [f:=(&&),g:=True,a:=True,as:=[False,True,True]]
|  | =f a (foldr f g as)[f:=(&&),g:=True,a:=True,as:=[False,True,True]]
|  | =(&&) True (foldr (&&) True [False,True,True]) % RHS substituted
|  |  |( &&) True (foldr (&&) True [False,True,True]) =: (&&) True b
                    ...match [b:=fold (&&) True [False,True,True]
|  |  | b[b:=foldr (&&) True [False,True,True]
|  |  | foldr (&&) True [False,True,True] =: foldr f g []
                   ...match fails
|  |  | foldr (&&) True [False,True,True] =: foldr f g (a:as)
                   ...match [f:=(&),g:=True,a:=False,as:=[True,True]
|  |  |  =(&&) False (foldr (&&) True [True,True])
|  |  |  | False && (foldr (&&) True [True,True])=:True && b
                    ...match fails
|  |  |  | False && (foldr (&&) True [True,True])=:_ && _
                    ...match []
|  |  |   |False
|  |  |False
|  |False
|False
```

Notice again that the end of the list, after the first `False` is encountered, never gets evaluated. Evaluation of the matching, which is sequential, only causes the evaluation of the second argument when the first argument of `(&&)` is `True`. If, however, we were to replace `(&&)` with `myand2`, above, then both arguments would have to be evaluated in order to resolve the patterns. This will produce a subtle change in the behaviour of the code would imply that `and [True,False,bottom]` would no longer terminate.

It is important to realize that pattern matching forces evaluation of arguments just sufficiently to determine whether the pattern matches. It does this sequentially from left to right across the pattern so that once a failure of a match is detected the later arguments never get evaluated – even if they have non-variable patterns which must be matched.

Notice also that if we had chosen to evaluate the arguments of a function before the evaluating the function itself – this is called the "by value" evaluation strategy and is, in fact, the evaluation strategy of choice in most programming languages – the behaviour will change. Thus, in `False && (fold (&&) True [True,True])` if we had evaluated the arguments of `(&&)` before evaluating the `(&&)`, this would have cause `fold (&&) True [True,True]` to be evaluated and we no longer would have had such an efficient program.

The evaluation strategy of Haskell is called "lazy evaluation". This means it only evaluates arguments of functions when they are needed and from left to right – this is called an "outermost leftmost" evaluation or a "by need" evaluation strategy – and, in addition, it "shares" evaluations of subexpressions. Lazy evaluation is a (provably) efficient evaluation strategy.

5 Hand evaluating anonymous functions

Often Haskell programs use "anonymous functions" or λ-abstractions. An example is given by the definition of `ord_list` as:

```
ord_inlist:: Ord a => a -> [a] -> Bool
ord_inlist x = foldr (\a b -> a==x || (x>=a && b)) False
```

The λ-abstraction is `(\a b -> a==x || (x>=a && b))`. It is called an anonymous function because we could equivalent define `ord_list` by replacing the anonymous function by a helper function:

```
ord_inlist:: Ord a => a -> [a] -> Bool
ord_inlist x = foldr helper False
  where
     helper a b = a==x || (x>=a && b))
```

Clearly, using an anonymous function – or λ-abstractions – can result in more succinct code ...

λ-abstractions are are evaluated by substitution: $(\backslash x \to t)\ s = t[x := s]$. There are some subtle issues when one is doing a substitution into a term which has bound variables (here x, in $(\backslash x \to t)$, is a bound variable in t) as one must avoid "variable capture". Variable capture occurs when the term is to be substituted in a position in which one of its free variables is also the name of a bound variable. As the name of the bound variables can be changed the solution is to rename the bound variable away from free variables.

Consider the example of evaluating `ord_inlist`. When one is searching for an element in an ordered list one can conclude the element is not present when the next element being examined is strictly larger than the element being sought. The definition of `ord_inlist`, as above, using a λ-abstraction is as follows:

```
ord_inlist:: Ord a => a -> [a] -> Bool
ord_inlist x = foldr (\a b -> a==x || (x>=a && b)) False
```

where (as in the Prelude) we shall assume:

```
(||) :: Bool->Bool->Bool
(||) False  b = b
(||) _  _ = True

(&&) :: Bool->Bool->Bool
(&&) True b = b
(&&) _  _ = False
```

We wish to show from a hand evaluation that the code has the correct behaviour of not examining parts of the list where the value exceeds the sought element.

```
ord_inlist 2 [1,4,8,11,22] := ord_inlist x [1,4,8,11,22]
      ... match [x:=2]
= foldr (\a b->a==x||(x>=a && b)) False [1,4,8,11,22] [x:=2]
= foldr (\a b->a==2||(2>=a && b)) False [1,4,8,11,22]
       := foldr f b []    ....fails
       := foldr f b (x:xs)
             ... match [f:= (\a b->a==2||(2>=a && b))
                    ,b:=False,x:=1,xs:=[4,8,11,22]]
= f x (foldr f b xs) [f:= (\a b->a==2||(2>=a&&b))
                     ,b:=False,x:=1,xs:=[4,8,11,22]]
= (\a b -> a==2 || (2>=a && b)) 1
               (foldr (\a b->a==2||(2>=a&&b)) False [4,8,11,22])
            -- substitute anonymous function
= 1==2||(2>=1&&foldr (\a b->a==2||(2>=a&&b)) False [4,8,11,22]))
      =: (||) False b
 | 1== 2  ... forces evaluation
 | False
 ... match [b:=2>=1&&foldr(\a b->a==2||(2>=a&&b))False [4,8,11,22])]
= b[b:=2>=1&&foldr (\a b->a==2||(2>=a&&b)) False [4,8,11,22])]
= 2>=1&&foldr (\a b->a==2||(2>=a&&b)) False [4,8,11,22])
       =: (&&) True b
 |  2>=1  ... forces evaluation
 |  True
 ... match [b:= foldr (\a b->a==2||(2>=a&&b)) False [4,8,11,22])]
```

```
= b [b:=foldr (\a b->a==2||(2>=a&&b)) False [4,8,11,22])]
=  foldr (\a b->a==2| (2>=a&&b)) False [4,8,11,22])
       := foldr f b []    ....fails
       := foldr f b (x:xs)
             ... match [f:= (\a b->a==2||(2>=a&&b))
                       ,b:=False,x:=4,xs:=[8,11,22]]
= f x (foldr f b xs)[f:= (\a b->a==2||(2>=a&&b))
                    ,b:=False,x:=4,xs:=[8,11,22]]
= (\a b->a==2||(2>=a&&b))4(foldr(\a b->a==2||(2>=a&&b))False [8,11,22])
           -- substitute anonymous function
= 4==2||(2>=4&&foldr (\a b->a==2||(2>=a&&b)) False [8,11,22]))
      := (||) False b
| 4==2 ... forces evaluation
| False
 ... match [b:=2>=4&&foldr(\a b->a==2||(2>=a&&b))False [8,11,22]]
= b[b:=2>=4&&foldr (\a b->a==2||(2>=a&&b)) False [8,11,22]]
= 2>=4 && foldr (\a b->a==2||(2>=a&&b)) False [8,11,22]]
   =: (&&) True b
 | 2>=4  forces evaluation
 | False  ... match fails
   =:  (&&) _ _ ... match  []
 = False
```

Again, one can check this is really what Haskell does by putting a `bottom` in the list after the `4`!

6 Hand evaluating `case` statements

Often one wishes to patten match the output of a function with in the body of a function: this is permitted by using a `case` statement. Here is a typical example of some code which uses a `case`:

```
data SF a = SS a | FF
deriving (Eq,Show)

 append :: [a] -> [a] -> [a]
 append xs ys = foldr (:) ys xs

 second:: [a] -> [a] -> SF  a
 second xs ys = case append xs ys of
                                  z:(x:_) -> SS x
                                  _ -> FF
```

This code extracts the second element from two lists which have been appended together. Let us hand evaluate `second [1,2,3] [4,5,6]`

```
second [1,2,3] [4,5,6]
 =: second xs ys   succeeds [xs:=[1,2,3],ys:=[4,5,6]]
 = case append xs ys of
             _:(x:_) -> SS x
             _ -> FF      [xs:=[1,2,3],ys:=[4,5,6]]
 = case append [1,2,3] [4,5,6] of
             _:(x:_) -> SS x
             _ -> FF
    | append [1,2,3] [4,5,6] =: z:(x:_)
                  (matching problem forces evaluation of append)
    | append [1,2,3] [4,5,6] =: append xs ys
                  ...succeeds [xs:=[1,2,3],ys:=[4,5,6]]
    | = foldr (:) ys xs [xs:=[1,2,3],ys:=[4,5,6]]
    | = foldr (:) [4,5,6] [1,2,3]
                =: foldr f b [] ...fails
                =: foldr f b (x:xs)
                ... succeeds [f:=(:),b:=[4,5,6],x:=1, xs:=[2,3]]
    | = f x (foldr f b (x:xs)) [f:=(:),b:=[4,5,6],x:=1, xs:=[2,3]]
    | = (:) 1 (foldr (:) [4,5,6] [2,3]) =: z:(x:_) [z:= 1]
                (composite pattern partially matches)
    |         |  foldr (:) [4,5,6] [2,3] =: (x:_)
                (forces evaluation of foldr ...)
    |         |  foldr (:) [4,5,6] [2,3] =: foldr f b []   ...fails
    |         |  foldr (:) [4,5,6] [2,3] =: foldr f b (x:xs)
                ...succeeds [f:=(:),b:=[4,5,6],x:=2,xs:=[3]]
    |         |  = f x (foldr f b xs)[f:=(:),b:=[4,5,6],x:=2,xs:=[3]]
    |         |  = (:) 2 (foldr (:) [4,5,6] [3]) := x:_
                 ... succeeds  [x:=2]
                      (underscore does not generate substitution)
    | append [1,2,3] [4,5,6] =: z:(x:_)
                 ...succeeds [z:=1,x:=2]
                      (original matching problem!)
  = SS x [z:=1,x:=2]
  = SS 2
```

Notice in this evaluation how the case statement generates sequentially two matching problems first `append [1,2,3] [4,5,6] =: z:(x:_)` and (if that one had failed) `append [1,2,3] [4,5,6] =: _`. The composite pattern `z:(x:_` forces a partial evaluation and generates a partial match (to get the substitution for `z`) before further forcing evaluation to obtain a substitution for `x`.

7 Hand evaluating tuples

Given the matching requirement to a tuple $g\ z =: (x, y)$, where $g\ z$ is not (immediately) a tuple, it may be supposed that $g\ z$ must be evaluated until a tuple structure appears at the outermost level.

However, this is not the case. Instead, Haskell interprets the $g\ z$ as a tuple without evaluating the arguments: thus, $g\ z = (\#1(g\ z), \#2(g\ z))$ and so allows the substitution $x := \#1(g\ z)$ and $y := \#2(g\ z)$. The "destructors", $\#1$ or $\#2$ applied to a term, however, do cause the term to be evaluated until a tuple is produced and the appropriate "destruction" step of selecting a component can be performed.

In evaluating tuples the other aspect of Haskell's evaluation – namely, sharing common subexpressions – also becomes more important: this avoids duplicate evaluations and tuples allow repeated computations (by duplication $x \mapsto (x, x)$).

Consider the following code for looking for the first repetition in a list:

```
snd:: (a,b) -> a
snd (_,x) = x

repeat:: Eq a => [a] ->  SF a
repeat ys = snd (foldr bump (FF, FF) ys)

bump:: Eq a => a -> (SF a,SF a) -> (SF a,SF a)
bump a  (x,y) =
 (SS a,  case x of
           SS b -> if a==b then SS a else y
           FF ->  y)

repeat'':: Eq a => [a] ->  SF a
repeat' ys = snd (foldr bump' (FF, FF) ys)

bump' a (FF,_) = (SS a,FF)
bump' a (SS b,y)| a == b = (SS a,SS a)
                | otherwise = (SS a,y)
```

`repeat` and `repeat'` are two different ways of writing a test to determine the first repetition in a list and to report which element is the first repeated (if there is one). Rather subtly the two programs have different behaviours although as functions they are completely equivalent on finite lists. Consider the evaluation of `repeat`:

```
repeat [1,1,2,2,3]
  =: repeat ys
         ... succeeds [ys := [1,1,2,2,3]]
  = snd (foldr bump (FF, FF) ys) [ys := [1,1,2,2,3]]
  = snd (foldr bump (FF, FF) [1,1,2,2,3])
   | := snd (_,x)
             ...succeeds [x:=#2 foldr bump (FF,FF) [1,1,2,2,3])]
   |  =  #2 (foldr bump (FF, FF) [1,1,2,2,3]) ...force evaluation...
   |  |   foldr bump (FF, FF) [1,1,2,2,3]?
         =: foldr f b [] = b  ... fail
         =: foldr f b (x:xs) ... succeeds
```

```
|  |  = f x (foldr f b xs) [f:=bump,b=(FF,FF),x:=1,xs:=[1,2,2,3]]
|  |  = bump 1 (foldr bump (FF,FF) [1,2,2,3])
       =: bump a  (x,y) ... succeed...
        [a:= 1, x:= #1(foldr bump (FF,FF) [1,2,2,3])
        ,y:= #2(foldr bump (FF,FF) [1,2,2,3])]
|  |  = (SS a,  case x of SS b -> if a==b then SS a
                                  else y; FF ->  y)
         [a:=1, x:= #1(foldr bump (FF,FF) [1,2,2,3])
         ,y:= #2 (foldr bump (FF,FF) [1,2,2,3])]
|  |  = (SS 1,  case #1  (foldr bump (FF,FF) [1,2,2,3]) of
                   SS b->if a==b then SS a
                         else #2(foldr bump (FF,FF) [1,2,2,3])
                   FF -> #2 (foldr bump (FF,FF) [1,2,2,3]) )
|  = case #1(foldr bump (FF,FF) [1,2,2,3]) of
                   SS b ->if a==b then SS a
                          else #2 (foldr bump (FF,FF) [1,2,2,3])
                   FF ->  #2 (foldr bump (FF,FF) [1,2,2,3]))
         ... forces evaluation ...
 |  |  =  #1 foldr bump (FF,FF) [1,2,2,3]) .. forces evaluation
 |  |  | = foldr bump (FF,FF) [1,2,2,3])
               =: foldr f b [] = b  ... fail
               =:  foldr f b (x:xs) ... succeeds...
                     [f:=bump,b=(FF,FF),x:=1,xs:[2,2,3]]
 |  |  |  = f x (foldr f b xs)[f:=bump,b=(FF,FF),x:=1,xs:[2,2,3]]
 |  |  |  = bump 1 (foldr bump (FF,FF) [2,2,3])
               =: bump a  (x,y) ...succeeds...
               [a:= 1, x:= #1  (foldr bump (FF,FF) [1,2,2,3])
               ,y:= #2 (foldr bump (FF,FF) [1,2,2,3])]
 |  |  |   = (SS 1, ...)
 |  |  = SS 1
 | = case SS 1 of  SS b -> if 1==b then SS 1
                         else #2(foldr bump (FF,FF) [1,2,2,3])]
                FF ->  #2(foldr bump (FF,FF) [1,2,2,3])])
 | = if 1==1 then SS 1
       else #2(foldr bump (FF,FF) [1,2,2,3])]
 = SS 1
```

Note how this exhibits a cut-off behaviour: the end of the list is never examined. However, `repeat'` does not exhibit this cut-off behaviour ... as you can check in Haskell by inserting a bottom after the first repetition – or indeed by doing a hand evaluation! This illustrates some fairly subtle aspects to the evaluation ...