

CM20167  
Topic 12: Haskell  
Introducing typed, lazy functional programming

Guy McCusker

1W2.1

## Introducing Haskell

Haskell, named after Haskell B. Curry, is a typed, lazy functional programming language. It is different from Lisp in several major ways.

- ▶ it is *typed*: its roots are in the typed  $\lambda$ -calculus rather than the untyped one; in fact Haskell's type system is significantly more sophisticated than that of the simply-typed  $\lambda$ -calculus
- ▶ it is *lazy*: evaluation is essentially normal-order, rather than applicative-order. The technique of sharing redexes using pointers is also used.
- ▶ it emphasizes *curried functions* over multi-argument functions.
- ▶ there are far fewer parentheses.

## Basic odds and ends

The basic types that we will look at are

- ▶ `Integer`—the type of integer numbers like 1 and -45.
- ▶ `Bool`—the type of truth values, called `True` and `False` in Haskell
- ▶ Function types, which are things like `Integer -> Bool`
- ▶ List types, written `[Integer]`. This is the type of lists that contain integers.

## Function types

The function types, which look like  $t_1 \rightarrow t_2$ , are very much like the types of the typed  $\lambda$ -calculus.

$t_1 \rightarrow t_2$  is the type of a function which can be given arguments of type  $t_1$ , and will return things of type  $t_2$ .

For instance, `not` is a function which negates truth values, so its type is

$$\text{Bool} \rightarrow \text{Bool}.$$

As in the typed  $\lambda$ -calculus, multi-argument functions are often written in curried form, so the type of an "and" or "or" operator would be

$$\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}.$$

## List types

In Lisp, we can put any pieces of data together into a list, e.g.

$$('a + 'b 'banana 13)$$

In Haskell, a list contains data of a specific type: the type `[Integer]` is the type of lists that contain only `Integer` data.

Lists are written with... gasp! ... square brackets and commas!

$$[1, 2, 3, 4, 5]$$

## Higher-order functions

Just as in Lisp, we can write functions that take other functions as arguments.

There are lots built in, including `map`, `filter` and `foldr`.

If  $f$  is a function of type `Integer -> Bool`, then `map f` is a function of type `[Integer] -> [Bool]`.

Thus the type of `map` is something like

$$(\text{Integer} \rightarrow \text{Bool}) \rightarrow [\text{Integer}] \rightarrow [\text{Bool}]$$

## Some function definitions

Let's define a function of that type.

```
positive :: Integer -> Bool
positive n = (n >= 0)
```

The first line is the *type declaration*. The second line is the actual function definition.

Then let's fire up the Haskell interpreter (I'm using *Hugs*) and try it out.

## Using the interpreter

```
Main> :t positive
positive :: Integer -> Bool

Main> positive 3
True

Main> positive (-24)
False

Main> :t map positive
map positive :: [Integer] -> [Bool]

Main> map positive [1,-2,3,-4,5]
[True,False,True,False,True]
```

:t asks the interpreter what type something has

## Type polymorphism and type inference

What happens if we ask the interpreter for the type of `map`?

```
Main> :t map
map :: (a -> b) -> [a] -> [b]
```

This is a *polymorphic type*: the `a` and `b` stand for any types we like.

So if we apply `map` to the `positive` function, which has type `Integer -> Bool`, the `a` becomes `Integer` and the `b` becomes `Bool`.

This version of `map` therefore has type

$$(\text{Integer} \rightarrow \text{Bool}) \rightarrow [\text{Integer}] \rightarrow [\text{Bool}]$$

and so `map positive` has type

$$[\text{Integer}] \rightarrow [\text{Bool}]$$

as we expected.

## Type polymorphism and type inference

Because of polymorphism, we can of course apply `map` to other types of function: if we define

```
double :: Integer -> Integer
double n = 2 * n
```

then we get

```
Main> :t double
double :: Integer -> Integer
```

```
Main> :t map double
map double :: [Integer] -> [Integer]
```

```
Main> map double [1,2,3,4,5]
[2,4,6,8,10]
```

## Type polymorphism and type inference

Notice that although the Haskell system gives a very general type for `map` itself

- ▶ it is able to work out much more specific types for things like `map positive` and `map double`: this is *type inference*
- ▶ those types are not necessarily the same: different uses of `map` do indeed have different types.

## Type inference

Type inference also means that we don't have to declare the type of everything. Haskell will work out a (general) type for us if we don't.

If we define

```
implies x y = not x || y
```

we get

```
Main> :t implies
implies :: Bool -> Bool -> Bool
```

- ▶ Since `not` has type `Bool -> Bool`, Haskell can work out that `x` must have type `Bool`.
- ▶ Thus `implies` must have type `Bool -> something`
- ▶ That `something` must also be `Bool` because the disjunction operator `||` has return type `Bool`.

## Type classes

As well as polymorphism, Haskell's type system has another wrinkle: type classes. A class describes a range of types with features in common.

- ▶ Certain types are *numeric*, so you can add, subtract, multiply and so on with them
- ▶ Certain types are *ordered*, so you can use  $<$ ,  $<=$  etcetera on them.

If we omit the type declaration for `double`, then Haskell infers a more general type for it. If we define

```
double n = 2 * n
```

```
implies x y = not x || y
```

```
greater m n = m >= n
```

Haskell does not complain: it tries to infer a general type for each function.

## Type classes

```
Main> :t double
double :: Num a => a -> a
```

```
Main> :t greater
greater :: Ord a => a -> a -> Bool
```

```
Main> :t implies
implies :: Bool -> Bool -> Bool
```

This says that

- ▶ `double` has type `a -> a`, for any type `a` which is numeric
- ▶ `greater` has type `a -> a -> Bool`, for any type `a` which is ordered
- ▶ `implies` has type `Bool -> Bool -> Bool` — a completely specific type.

## Self-application

Haskell's type inference algorithm can confirm for us what we worked out about the typed  $\lambda$ -calculus: the self application `x x` is not typeable.

If we try to define

```
selfapply x = x x
```

Haskell complains:

```
*** Expression      : x x
*** Term            : x
*** Type            : a -> b
*** Does not match : a
*** Because         : unification would give infinite type
```

which is pretty much what we concluded last time.

## Laziness

Haskell is a *lazy*, aka *non-strict* language.

In  $\lambda$ -calculus terms this means that normal-order reduction is used, rather than applicative-order as in Lisp.

### Definitions

```
loop = loop  
  
noprobem x y = x
```

### Interpreter

```
Main> :t loop  
loop :: a  
  
Main> :t noprobem  
noprobem :: a -> b -> a  
  
Main> noprobem 3 loop  
3
```

Note that `loop` is a value of *any type* at all!

## Lazy lists

But there is much more to laziness in Haskell than this. In particular, the list constructor is lazy.

In Haskell, lists are constructed with `:`, as you can see.

```
Main> 1 : 2 : 3 : 4 : []  
[1,2,3,4]
```

The colon is the equivalent of Lisp's `cons` (for lists, not any pairs), and `[]` is the equivalent of Lisp's empty list, `nil`.

The exciting part is that `:` is non-strict in its second argument.

## Non-strict cons

If we evaluate `loop`, the interpreter will go into an infinite loop, and never give us an answer.

But if we define

```
lazylist = 1 : loop
```

the interpreter is happy, and we can even investigate elements of this list.

```
Main> :t lazylist  
lazylist :: [Integer]
```

```
Main> head lazylist  
1
```

head takes the first element of a list, a bit like `car`

Asking for other elements of the list will of course result in a loop.

## "Infinite" lists

Because the cons operator is non-strict, we can form *partial* lists like the `lazylist` example above.

In a partial list, some of the elements have been evaluated, and some of them have not. Roughly speaking, when a list is built using `:`, the head is evaluated but the tail is not. Only when we actually probe into the tail is it actually evaluated.

With the aid of recursion, those as yet unevaluated elements could go on forever.

## "Infinite" lists

### Definitions

```
longlist = 1 : longlist
```

tail takes the rest of a list (all but the head), a bit like `cdr`

take takes the first few elements of a list

### Interpreter

```
Main> :t longlist
longlist :: [Integer]
```

```
Main> head longlist
1
```

```
Main> head (tail longlist)
1
```

```
Main> take 10 longlist
[1,1,1,1,1,1,1,1,1,1]
```

## Amazing tricks with infinite lists

Suppose we want to compute the Fibonacci numbers. We could write a recursive definition, like this:

```
fibb n = if (n == 0) then 0 else
         if (n == 1) then 1 else
         fibb (n-1) + fibb (n-2)
```

or we could use Haskell's pattern matching syntax, like this:

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib n + fib (n+1)
```

(More on pattern matching later)

Both of these work, but they take a very long time when computing even the 30th Fibonacci number.

## Amazing tricks with infinite lists

The reason they take so long is that they keep repeating work: for example, in computing the 30th Fibonacci number,

- ▶ one recursive call computes the 29th, which in turn computes the 28th and 27th
- ▶ another recursive call computes the 28th, which in turn compute the 27th and 26th

and so on.

What is happening here is that the results of previous recursive calls are being repeatedly recalculated, rather than remembered and reused.

If we instead compute the *infinite list of Fibonacci numbers*, we can use the results we've computed more than once.

## Amazing tricks with infinite lists

The function `zipWith` is similar to `map`, but it takes three arguments: a function and *two* lists.

It computes a new list by applying the function to *two* arguments, one from each list, taking them in order. So e.g.

```
Main> zipWith (+) [1,2,3] [20,30,40]
[21,32,43]
```

We can now compute all the Fibonacci numbers like this:

```
fibn = 0 : 1 : zipWith (+) fibn (tail fibn)
```

## Eratosthenes' sieve again

Remember the sieve function we wrote to compute primes?

```
(defun indivisible (m) (
  lambda (n)
    (not (zerop (% n m)))
  )
)

(defun sieve (l)
  (if (null l)
      ()
      (cons (car l) (sieve (filter (indivisible (car l)) (cdr l))))
  )
)
```

## Eratosthenes' sieve again

Here it is in Haskell:

```
sieve [] = []  
sieve (x:xs) = x : sieve (filter (\n -> (mod n x > 0)) xs)
```

This uses *pattern matching* to do two things:

- ▶ separate out the case of the empty list and the case of a non-empty list into two lines
- ▶ in the non-empty case, get hold of the head and tail of the input list in a low-overhead syntax

It also uses Haskell's  $\lambda n \rightarrow$  something syntax (read as  $\lambda n$ .something) to define an anonymous function along the way.

## Pattern matching

To use pattern matching, we write our function definition over several lines.

Each line defines what the function does on input of a certain shape. Very common patterns are

- ▶ for lists, the empty list in one line and  $x:xs$  in the next line; this pattern matches any non-empty list, and  $x$  gets bound to the head, while  $xs$  gets bound to the tail
- ▶ for numbers, zero in one line and  $n+1$  in the next line; the pattern  $n+1$  matches any integer greater than 0, and binds  $n$  to that integer's predecessor.

Other patterns are allowed too, like  $(n+2)$  for numbers or  $x:y:xs$  for lists, for example.

When evaluating a function application, Haskell looks at the lines of the definition in order and takes the first one which matches the argument.

## Computing primes

We can use the sieve to compute a lot of primes. An infinite list, in fact!

In Lisp we had to supply it with some finite list like

```
(2 3 4 5 6 7 8 9 ... 1000)
```

Haskell allows us to write `[2..]` for the infinite list

```
[2,3,4,5,6,7,8,9,10...]
```

and then we can execute

```
Main> sieve [2..]  
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,  
73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,  
157,163,167,173,179,181,191,193,197,199,211,223,227,229,233,  
239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,  
331,337, ^C{Interrupted!}]
```

to compute primes until we hit ctrl-C (or the stack overflows).

## More infinite lists

We didn't need to use the built-in `[2..]`: we could have defined, for example

```
listofthings = 2 : map (+1) listofthings
```

Here's the infinite list of factorial numbers:

```
facts = 1 : zipWith (*) facts [1..]
```

Here are all the squares:

```
squares = map (^2) [1..]
```

## Just a taste...

That is just a taste of what can be done in Haskell.

It has a great many interesting features, some of which are unique, e.g. *monads* which are Haskell's way of adding, but controlling, side-effects.

It is also surprisingly fast to execute, particularly if you use a compiler rather than an interpreter.

Go to <http://www.haskell.org> to find out more, and if you'd like to play with an implementation, consider Hugs, an easy to use interpreter, or the Glasgow Haskell Compiler (`ghc`), a full scale compiler.