Introduction to Haskell

Frederick Ross and lots of folk from #haskell

April 26, 2008

1 Preliminaries

Source code in Haskell is organized into modules. In today's environments, modules correspond to files, but this isn't fixed in stone. Modules contain definitions, visible only within the module and in other modules that explicitly import them. Two modules can both define functions named *foo* without collision.

The language is case sensitive. Text between `--` and the end of the line is a comment.

Outside of Haskell, programs are typically a sequence of instructions to the computer. Take the program

```
x := 3;
print x;
x := 5;
```

Tag each line with its position in the sequence

```
(0) x := 3;
(1) print x;
(2) x := 4;
```

We can rearrange the statements as much as we want, if we agree that statements with lower numbers are executed before statements with higher numbers. So

```
(2) x := 4;
(0) x := 3;
(1) print x;
```

is identical to

```
(1) print x;
(2) x := 4;
(0) x := 3;
```

Now let's cut off the numbers. The immediate response is "why would you do something so ridiculous?" One processor executes instructions one after another. But keeping those numbers straight across several processors or several computers is hard. Carefully building locks and synchronization points are necessary to limit the damage.

Or cut off the line numbers and see what is still possible. This is how Haskell works. We cede control of when — or if — statements are executed to the compiler. Only one ordering remains: in `sqrt(cos(3))`, the subexpression `cos(3)` will be executed first. The order of statements in Haskell modules is immaterial, with two exceptions:

1. Since files correspond to modules, every file's first line of code must be of the form

```
module Name where
```

Here `Name` is the name of the module. Module names must be capitalized, and bear the same name as the file that contains them. `Bar` must be in `Bar.hs` or the compiler won't be able to find it.

2. One module can import the definitions from another. `Foo` can use the definitions in `Bar` by putting the command

   ```
   import Bar
   ```

   after its module declaration. It can import another module with a prefix

   ```
   import qualified Bar as B
   ```

   so all functions in Bar are called `B.`*function* in `Foo`. All import lines must come after the module declaration and before any other code.

After this, definitions can come in any order.

You can compile Haskell code to a standalone executable, or load modules into an interpreter to exercise their functionality. Only a module `Main` containing function *main* can be compiled to an executable. For example, we compile the module

```
module Main where
main = putStrLn $ show 3
```

with

```
$ ghc --make Main -o prog
```

which produces an executable `prog` that we can run like any other program. `--make` forces the compiler to correctly compile and link all modules which `Main` depends on.

Interpreters accept commands in order, so how do they mesh with Haskell? We can evaluate expressions, but definitions have a special syntax. The minimum is to load your interpreter (`ghci`, for example), and in it load your module

```
:l Main
```

Run *main* by typing its name at the prompt. `ghci` is capable of more, but this is enough for now.

2 Expressions & Definitions

All Haskell code is expression and definition. Expressions are strings for the compiler to evaluate.

```
(2*3) + 5
"First" ++ "Second"
True && False
```

Every expression has unique and unchanging type, written *expression* `::` *Type*.

```
3 :: Int
5.24e6 :: Double
"The quick brown fox" :: String
'e' :: Char
True :: Bool
False :: Bool
```

The names of types, their constructors (*True* and *False* for `Bool`), and modules begin with a capital letter. Nothing else in the language can.

`Int` a machine integer; `Integer` is an arbitrary precision integer. `Float` exists, but all effort, hardware, and algorithms focus on `Double`. Use it instead. Numbers have all the operations we expect: `+`, `-`, `*`, `/`, `<`, `>`, `<=` (at most), `>=` (at least), `==`, `/=` (not-equals), *max*, and *min*. Prefix functions like *max* and *min* take no parentheses around their arguments, nor commas seprating them, just spaces.

Statements herein like *expr1* `==` *expr2* evaluate to `True` in Haskell: `expr1` evaluates to `expr2` or an identity holds among the expressions involved.

```
3 < 5 == True
max 9.5 2.1 == 9.5
```

Single characters are delimited by single quotes (`' '`), strings by double quotes (`" "`). For historical reasons, Haskell has two kinds of strings, `String` and `ByteString`.

`String` is the native representation, handles Unicode, and is a factor of ten slower than `ByteString`, which handles only ASCII. `ByteString`'s speed is comparable to C. If you are handling large quantities of ASCII text, use ByteStrings. You must add

```
import qualified Data.ByteString.Lazy.Char8 as B
```

to the top of your module, and functions on the ByteStrings as `B.`*function*. *pack* and *unpack* turn `String`s into `ByteString`s and vice versa.

```
unpack (pack "Test") == "Test"
```

Values and types displayed by Haskell are Haskell code. For most types, but not functions, data is read, written, and represented identically. `show` produces a string corresponding to a value, and `read` turns it back into a value.

show `::`
`a -> String`
read `::`
`String -> a`

```
read (show 23.552) == 23.552
```

Definitions take the form *name* `=` *expression*.

```
a = 3
b = 5 * (2+4)
```

You can split long definitions across lines by indenting lines after the first. They need not be indented the same, only more than the first.

```
foo = max
   3
 2
bar = "A very long string which I will break " ++
      "across lines."
```

3 Functions

Functions are the same as strings or integers: they are values with literal representations and types. Their literal form is

\arg1 arg2 ... argn `->` *expression*

where *expression* involves any of *arg1...argn*.

```
\x -> 3 + x :: Integer -> Integer
\y z -> y * z :: Double -> Double -> Double
\s1 s2 -> s1 ++ s2 :: String -> String -> String
```

They can be the expression in a definition.

```
a = \x -> 3+x
c = \s1 s2 -> s1 ++ s2
```

The types look bizarre. `String -> String -> String` should be a function from two strings to another string, but it can be read as `String -> (String -> String)`, a function from strings to another function. In truth, it's both. The syntax for using functions makes this obvious.

```
a 4 == 7
c "A " "string" == "A string"
```

No parentheses, no commas, just a space separated list of arguments. If we give *c* one argument instead of two,

```
d = c "A " :: String -> String
```

we get another function. Function application is still an expression. We can layer it, as in

```
a (2 * (a 3)) == 15
d (d "rat") == "A A rat"
```

The last line must be `d (d "rat")`, not `d d "rat"`. A sequence of function applications binds left: `x y z q == ((x y) z) q`. `d d "rat" == (d d) "rat"`, but `d` is of type `String -> String` — it takes only arguments of type `String`, not `String -> String`. Mismatched types won't even compile. A type system this strong changes the way you work. If it compiles, it's probably right. The first thing to know about a function is its type.

Haskell provides several syntactic shortcuts function definitions. The most common is

function arg1 arg ... argn = *expression*

```
a x = 3 + x
b y z = y * z
c s1 s2 = s1 ++ s2
```

Infix operators — functions which sit between their arguments, such as `+`, ***, and *++* — are normal functions with peculiar syntax. We make any infix function prefix by surrounding it with parentheses (`( )`), any prefix function infix by surrounding it with backquotes (`` ` ` ``). Thus all the following expressions evaluate to true:

```
3 + 5 == 8
(+) 3 5 == 8
c "A " "rat" == "A rat"
"A " `c` "rat" == "A rat"
```

Any function used infix is applied to two arguments, so one argument functions do not work. Functions of five arguments used infix simply evaluate to other functions.

```
g x y z q = (x+y) * (z+q)
3 `g` 5 :: Integer -> Integer -> Integer
```

We can define infix functions as if they were prefix

```
(^^) = \x y -> x + (2*y)
```

or directly as infix

```
x ^^ y = x + (2*y)
```

The syntactic shortcut for defining functions does more. Distinct constructors for a data type (`True` and `False` for `Bool`) can be defined separately.

```
toBit True = 1
toBit False = 0
```

This works with specific values of any time. For instance

```
f 3 x = x*x
f 5 x = 2*x
```

In these cases the order matters! The statements are checked from top to bottom, and the first one to match is the last one examined.

```
f y x = 3*x
f 3 x = 4*x
```

always evaluates to `3*x`; the second line is never used.

4 Common Data Structures

Three data structures carry the brunt of labor in Haskell: lists, tuples, and `Maybe`. `Maybe` must be imported at the top of your module with

```
import Data.Maybe
```

`Maybe` handles values which may be undefined. It has two constructors, *Just x* and *Nothing*, and is used as

```
Just 3 :: Maybe Integer
Just "A string" :: Maybe String
Nothing :: Maybe a
```

What is `Maybe a`? Haskell imposes the weakest possible type on expressions, including functions which accept classes of types instead of just one. `Nothing` applies equally well to any type, so Haskell inserts a variable `a` which is set appropriately when the constructor is used. You can define restricted classes of types with specific properties and write functions that operate on that class, but this introduction won't discuss this further.

The syntactic shortcut for defining functions extends naturally to

```
addTwo :: Maybe Int -> Maybe Int
addTwo (Just x) = Just (x+2)
addTwo Nothing = Nothing
```

If `Just x` were not in parentheses, Haskell would read it as two argument to `addTwo`. `x+2` must be between parentheses or it would be interpreted as `(Just x) + 2`.

A tuple consists of a fixed number of slots of fixed types, for example

```
(2,"fox") :: (Int,String)
('e', "fox", "hen", 53e12) :: (Char, String, String, Double)
```

This syntax — parentheses surrounding a comma-separated sequence of expressions — doesn't permit a tuple with one slot, but these are never useful. Tuple patterns also in function definitions.

```
rootsOfQuadratic (a,b,c) = ( (-b + sqrt(b*b - 4*a*c)) / (2*a),
                             (-b - sqrt(b*b - 4*a*c)) / (2*a) )
```

Often a tuple is an adequate representation of a small data structure. Haskell provides more sophisticated mechanisms, but this introduction will not cover them.

The last structure, lists, are central to Haskell. Lists are sequences of arbitrary length but fixed type. Specifically, they are one directional linked lists, which defines the natural operations on them. Their literal form is a comma separated sequence surrounded by square brackets (`[ ]`). `[]` is the empty list. You must handle this case when writing functions on lists.

```
[] :: [a]
[1,2,3] :: [Integer]
['a','b','c'] :: [Char]
["first", "second", "third"] :: [String]
[[1,2], [3,4], [5,6]] :: [[Integer]]
```

The last line shows nested lists. Lists need not be finite. We can take elements from the beginning despite having infinitely many others. Infinite structures are problematic in most languages, but denying ourselves statement order lets us consign to the compiler when, and if, to evaluate an expression. “Never” is acceptable, so we can work with infinite lists so long as we evaluate only a finite number of elements. The compiler will never evaluate the rest.

Haskell provides syntactic shortcuts for writing infinite lists. For numbers and characters we can write

```
[1..] :: [Integer]
['a'..] :: [Char]
```

The same notation also yields finite ranges.

```
[1..10] :: [Integer]
['a'..'q'] :: [Char]
```

The infix function `(:)` builds a list consisting of its first argument followed by its second argument. `head` returns the first element of a list, `tail` the rest. At all times,

(:) ::
`a -> [a] -> [a]`
head ::
`[a] -> a`
tail ::
`[a] -> [a]`
reverse ::
`[a] -> [a]`

```
\texttt{(head l) : (tail l) == l}
```

`reverse` reverses a list, but is computationally expensive. Avoid it by arranging your lists in the right direction initially.

`:` appears when we write functions on lists. We name as many initial elements we need, plus the tail of a list with

$$(a\texttt{:b:}\ldots\texttt{:}\,\mathit{tail})$$

```
sumOfFirstTwo (x1:x2:xs) = x1+x2

sum (y:ys) = y + sum ys
sum [] = 0
```

Lists arise naturally in many situations, but their most important role in Haskell is control flow. Haskell iterates through lists where most languages loop. Surpisingly, this is far simpler.

Strings are lists of characters. Any list operation works seamlessly on strings. The infix function $++$ concatenates both strings and lists.

```
['a','b','c'] == "abc"
"abc" ++ "def" == "abcdef"
[1,2,3] ++ [4,5,6] == [1,2,3,4,5,6]
```

`words` splits a string at whitespace boundaries, `lines` at newlines.

words ::
`String -> [String]`
lines ::
`String -> [String]`

```
words "The quick brown fox\njumped over the lazy dog."
    == ["The","quick","brown","fox","jumped","over","the","lazy","dog."]
lines "The quick brown fox\njumped over the lazy dog."
    == ["The quick brown fox","jumped over the lazy dog."]
```

ByteStrings have homologs of all list functions, but not *words* and *lines*. Use `split` in place of *lines*,

split ::
`Char8 -> ByteString -> [ByteString]`

```
map B.unpack $ B.split '\n'
   (B.pack "The quick brown fox\njumped over the lazy dog.")
    == ["The quick brown fox","jumped over the lazy dog."]
```

and `splitWith` for *words*. The module `Data.Char` defines character predicates such as `isSpace`.

splitWith ::
`(Char -> Bool) -> ByteString -> [ByteString]`
isSpace ::
`Char -> Bool`

```
splitWith Data.Char.isSpace
     (pack "The quick brown fox\njumped over the lazy dog.")
     == ["The","quick","brown","fox","jumped","over","the","lazy","dog."]
```

5 Control Structures

Programming without repetition and branching is pointless. The shortcut syntax for functions gives us one way to branching, and Haskell has others. The next most common is `if...then...else`.

```
isNegative x = if x<0 then True else False
```

`if...then...else` is a syntactic shortcut for a normal function of type `Bool -> a -> a -> a`. We can define it as

```
cond True x _ = x
cond False _ x = x
```

Underscores (_) stand for an argument we won't use in the expression. They are placeholders. The variables in each version of the function are independent, but for readability you should name your variables consistently across versions.

`cond` is equivalent to `if...then...else`. In particular, you must always have an else condition. Otherwise the construct would not be an expression.

How do we iterate? Iteration is a combination of three forms: perform an identical action on a set of things; conditionally perform an action on a set; or we sequentially perform actions that carry over from one iteration to the next.

Typically adding 3 to every element of a list takes the form

```
for i = 1 to 5
    x[i] := x[i] + 3;
end
```

Define a function *f* which adds three to its argument, and rewrite the loop as

```
for i = 1 to 5
    f(x[i]);
end
```

Abstract the loop to a function, conventionally called `map`.

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

```
map (3+) [3,2,7,5] == [6,5,10,8]
```

(3+) is another syntactic shortcut. When we make an infix function prefix by surrounding it with parentheses, we can give it either argument to get a one argument function.

Iterating while conditionally performing an operation divides into two functions: one iteration which selects the elements to operate on, one iteration to perform the operation. The latter uses *map* again. The former uses `filter`. *filter*'s first argument is a boolean-valued function. *filter* returns those elements of its second argument on which the first argument evaluates to `True`.

filter ::
`(a -> Bool) -> [a] -> [a]`

```
filter (<0) [5,-2,-3,7,9,-1] == [-2,-3,-1]
```

There is a problem: we iterate once with *filter*, and once with *map*, twice the required number. Actually, as the compiler evaluates functions when it wishes, it runs both conditional and transformation during one iteration. Layering these functions adds no additional iterations.

We often use the index variable's value in a loop.

```
for i = 1 to 5
    x[i] := x[i] + i;
end
```

The index is not available in *map* or *filter*. First rewrite the loop.

```
idx = {1,2,3,4,5};
for i = 1 to 5
    x[i] := x[i] + idx[i];
end
```

This form doesn't depend on the index variable, but we can only iterate over one list at a time. One more rewrite finishes it.

```
idx = {1,2,3,4,5};
for i = 1 to 5
    y[i] := (x[i], idx[i])
end
for i = 1 to 5
    y[i] := (first(y[i]) + second(y[i]), second(y[i]))
end
```

where `first` and `second` pick out those elements of our tuple. The second loop is a *map*. The first one is common enough to deserve a name: `zip`.

zip ::
`[a] -> [b] -> [(a, b)]`

zip fuses two lists item by item into a list of tuples; *zip3* does the same for three lists. Their inverses *unzip* and *unzip3* make lists of tuples into tuples of lists.

```
zip [1,2,3] ['a','b','c'] == [(1,'a'),(2,'b'),(3,'c')]
unzip [(1,'a'),(2,'b'),(3,'c')] == ([1,2,3],"abc")
```

`zip`'s arguments need not be the same length. It returns a list as long as the shortest of its arguments, and discards all later elements. With *zip* the loop becomes

```
(map (\(x,y) -> x+y) $ zip [3,2,7,5] [1..]) == [4,4,10,9]
```

($) is a useful operator in Haskell to save on parentheses. It obeys

```
a $ b c d ... = a (b c d ...)
```

We repeat *($)* instead of layering parentheses as we layer function calls. *($)* is a normal function defined by

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Combining *map* and *zip* is so common that we also give it a name: `zipWith`, and *zipWith3* for three arguments.

zipWith ::
`(a -> b -> c) -> [a] -> [b] -> [c]`

```
(zipWith f m n) == (map (\(x,y) -> f x y) $ zip m n)
```

The loop becomes

```
(zipWith (+) [3,2,7,5] [1..]) == [4,4,10,9]
```

Certain strange things become possible with these functions. Here are the Fibonacci numbers

```
(fibs = 0 : 1 : zipWith (+) fibs (tail fibs)) == [0,1,1,2,3,5,8,13,...]
```

The third loop pattern iterates, using the result of its previous iteration in its next.

```
y := 0;
for i = 1 to 5
    y := y + x[i];
end
```

If we write this out mathematically it looks like

$$(((((0 + x_1) + x_2) + x_3) + x_4) + x_5)$$

We give this a name, `foldl`. The first argument is the binary operation (+ in the expression above); the second is the starting value (0 in the expression above, the initializing line `y := 0` before that); the last is the list to fold.

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

It is called *foldl* instead of *fold* because we can group in the opposite direction

$$(x_1 + (x_2 + (x_3 + (x_4 + (x_5 + 0)))))$$

to get *foldr*. Prefer *foldr*, for it uses less stack space. *foldl* must reach the end of the list then work its way back. For finite lists,

```
foldl f z xs == foldr (\x y -> f y x) z (reverse xs)
```

`scanl` works like a fold, but returns all intermediate states as a list.

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

```
(scanl (+) 0 [1..10]) == [0,1,3,6,10,15,21,28,36,45,55]
```

foldr is the proper structure for database access. ByteStrings again have their own definitions of all these functions.

6 Files and Transput

Haskell's transput functions appear deceptively simple: *readFile*, *writeFile* and *appendFile* for files; *getLine*, *putStr* and *putStrLn* for the terminal. But their types...

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
getLine :: IO String
putStr :: String -> IO ()
putStrLn :: String -> IO ()
```

`()` is the nil type, with one corresponding value, `()`. But what is `IO`? How do we get past it to the strings underneath?

We threw away all concept of order and timing. What does

```
k := readFile "a_file"
writeFile "a_file" k
```

do? Functions are guarunteed to be evaluated after their arguments, and nothing else, but we enforce ordering with this: make functions depend on trivial evaluations, tokens passed from other functions, as well as real data. Each function doing transput returns its result plus such a token. `IO` hides this token.

None of the functions accept `IO`; they only produce them. Haskell provides other functions to link these together, correctly directing tokens and data. (>>) connects independent transput events.

```
(>>) ::
IO a -> IO b -> IO b
```

```
putStr "This " >> putStrLn "is a test."
```

`putStr "This " :: IO ()` returns a token in `IO ()`. *(>>)* forces `putStrLn "is a test." :: IO ()` to depend the token's evaluation.

(>>=) connects transput events which are not independent by pushing the value of its first argument to its second argument as well as resolving the tokens. To read the contents of a file, then write them to a new file

```
(>>=) ::
IO a -> (a -> IO b) -> IO b
```

```
readWrite infile outfile = readFile infile >>= writeFile outfile
```

(>>) and *(>>=)* only work on transput functions. To use them with pure functions, we lift the pure function into the world of transput. `return` lifts single values into transput. We lift functions by composing it with *return*.

return ::
`a -> IO a`

```
return . (+3) :: Integer -> IO Integer
```

(.) composes functions, so

```
(f . g) x == f (g x)
```

If we made return operate on functions instead of values, we would have to write a separate function for values. Composing functions is so easy that we save a name by only creating a function to lift values.

Transput works for arbitrarily large files. Sacrificing order allows the Haskell compiler to break transput into chunks as it sees fit.

We achieved transput only by restoring the line numbers to our code. Much of our code now has lovely properties, but do they justify the complication of transput?

They do. Like repetition, we build functions over *(>>)*, *(>>=)*, and *return* to make transput effortless. Also, transput is a value. We can manipulate it in tiny grains, or compose it into larger structures. When we need transput, we use a grain just large enough for the purpose, then fall back into orderless code. Distributed systems make this compelling: local transput has its own line numbers independent of all other transput. Replacing global progression of state in a program with local grains is a step akin to replacing `GOTO` with structured programming.

Like structured programming, we need constructs to make transput convenient. The library `Control.Monad` defines analogs of the repetition functions: *mapM*, *filterM*, *zipWithM*, *foldM*.

mapM is an extension of *(>>)*. To print each entry of a list,

```
printList x = mapM (putStrLn . show) x
```

`printList [3.5, 1.22, 3e2]` produces

```
3.5
1.22
300.0
```

filterM selects elements depending on transput. To let the user say yes or no to every element of a list, then print those selected,

filterM ::
`(a -> IO Bool) -> [a] -> m [a]`

```
userSelect x = putStr ((show x) ++ " (y/n)? ") >> getLine >>= return.(=="y")
queryList x = filterM userSelect x >>= printList
```

filterM's result is lifted into transput, since the decisions came from transput. Its result must be fed on with *(>>=)*. This is true for all the transput control structures.

```
printSquaresOfSelected x = filterM userSelect x
      >>= mapM (\x -> putStrLn (show (x*x)))
```

We can use *mapM* getting results from transput. To return answers to a list of questions posed to the user,

```
userResponse q = putStrLn q >> getLine
poseQuestions qs = mapM userResponse qs
```

zipWithM is the same as *mapM* but takes two lists which it fuses element by element.

zipWithM ::
`(a -> b -> IO c) -> [a] -> [b] -> m [c]`

```
zipWithM f a b == mapM (\(x,y) -> f x y) $ zip a b
```

foldM allows dependence. To print a list of strings, letting the user turn printing on and off by typing any character at the keyboard between lines,

foldM ::
`(a -> b -> IO a) -> a -> [b] -> IO a`

```
printAndToggle True s = putStrLn s >> getLine >>= return.(=="")
printAndToggle False s = getLine >>= return.(/="")

userIntersperse q = foldM printAndToggle True q
```

Two things are missing from this discussion of transput. The first is do notation, a transput syntax that resembles sequences of commands in other languages. You should avoid it until you are comfortable with transput in the notation here.

The second is abstracting the machinery that chains transput together. The resulting structure is called a "monad." In category theory, the branch of mathematics which gave birth to monads, they are connections between very different spaces that let you manipulate the far space from the near one: construct a space where programs have the properties you want — all functions log what they do; statements combine into backtracking searches *à la* Prolog; all calculations are actually probability distributions — and lift constructions from familiar Haskell into this space.

7 Command Line Arguments

Only handling commandline options now prevents us from writing simple shell utilities in Haskell. getArgs in System.Environment returns a list of the program's arguments. To insert line numbers in all files passed as arguments,

getArgs ::
`IO [String]`

```
insertNumbers ls = zipWith
      (\x y -> (show x ++ " " ++ y)) [1..] ls
processFile filename = readFile filename
      >>= return.insertNumbers.lines
      >>= mapM putStrLn
main = getArgs >>= mapM processFile
```

System.Environment also provides withArgs to test programs.

withArgs ::
`[String] -> IO a -> IO a`

```
withArgs ["file1","file2"] main
```

prints the contents of `file1` and `file2` with numbers prepended to each line.

This introduction handled commandline arguments and user interaction directly. In general you should use Haskell's versions of `getopt` and `readline` under `System.Console`. Learning enough to use these libraries is a good next exercise.