Functional Programming Lecture notes

Piyush P Kurur

October 31, 2011

Lecture 1

Introduction

In this course we will learn functional programming through haskell. The main motivation of this course is to prepare people for real world programming. To be a successful haskell programmer, I believe, one needs the uncluttered thought of a mathematician combined with the pragmatism of an Engineer. One can learn the basics in a matter of days. However, one requires regular practice to be an accomplished programmer.

1.1 What will be done in this course ?

Covering the entire language, the mathematical background and the libraries is impossible in a semester long course. So in the class we will learn the basics of haskell and some of the mathematical notions. In fact, most of the class will be devoted to understanding these mathematical concepts. Programming in haskell is a good exercise to your mathematical muscle and thus this is an important component of this course. But I hope this course does not end up being a collection of abstract nonsense. Therefore, we will have regular (i.e. weekly) *ungraded* assignments to pull us back to real world.

1.2 Evaluation.

Your grade will depend on your performance in exams (which constitutes 70% of the marks) and in the semester long project (which takes up the remaining 30% of marks). There will be zero credit for the weekly assignments. However, they are what will help you build the knowledge.

Projects can be done in a group of *3 or less*. Beside the projects have to be under version control and since it is expected to be in haskell, should be released as a cabal package. I recommend using darcs as the version control program. We will **NOT** discuss darcs or cabal in the course, you will have to learn it on your own. You may choose on what you want to work on but I will put up a list of sample projects for you to get an idea of the scope of haskell as an application development framework.

Besides, I encourage you to use haskell in your other programming needs like B.Tech project, M.Tech/Ph.D Thesis etc.

Lecture 2

Warming up

We will start with the haskell interpreter `ghci`. To start the interpreter type the following on the command line. You might see something like this.

```
$ ghci
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

The `Prelude`> is the prompt and the interpreter is expecting you to type in stuff. What you can type is any valid haskell expression. You can for example use ghci like a calculator.

```
Prelude> 42
42
Prelude> 2 + 4
6
Prelude> sqrt 2
1.4142135623730951
```

The line 5 `sqrt 2` is a *function application*, the function sqrt is applied on 2. In haskell, applying a function `f` on an argument `x` is done by placing `f` on the left of `x`. Function application associates to the left, i.e. `f x y z` means `((f x) y)z`.

2.1 Types

Although it did not appear to be the case in this limited interaction, Haskell is a strongly typed language unlike python or scheme. All values in haskell have a type. However we can almost always skip types. This is because the compiler/interpreter infers the correct type of the expression for us. You can find the type of an expression prefixing it with ‘:type’ at the prompt. For example

```
Prelude> :type 'a'
'a' :: Char
```

In haskell the a type of a value is asserted by using `::` (two colons). The interpreter just told us that `'a'` is an expression of type `Char`.

Some basic types in Haskell are given in the table below

2.2 Lists and Strings.

A list is one of the most important data structure in Haskell. A list is denoted by enclosing its components inside a square bracket.

```
Prelude> :type ['a','b']
['a','b'] :: [Char]
Prelude> :type "abcde"
"abcde" :: [Char]
```

A string in Haskell is just a list of characters. The syntax of a string is exactly as in say C with escapes etc. For example `"Hello world"` is a string.

Unlike in other languages like python or scheme, a list in Haskell can have only one type of element, i.e. `[1, 'a']` is *not* a valid expression in haskell. A list of type `t` is denoted by `[t]` in haskell. Example the type `String` and `[Char]` are same and denote strings in Haskell.

2.3 Functions.

In haskell functions are first class citizens. They are like any other values. Functions can take functions as arguments and return functions as values. A function type is denoted using the arrow notation, i.e. A function that takes an integer and returns a character is denoted by `Integer -> Char`.

```
Prelude> :type length
length :: [a] -> Int
```

Notice that the interpreter tells us that length is a function that takes the list of `a` and returns an `Int` (its length). The type `a` in this context is a *type variable*, i.e. as far as length is concerned it does not care what is the type of its list, it returns the length of it. In Haskell one can write such generic functions. This feature is called polymorphism. The compiler/interpreter will appropriately infer the type depending on its arguments

```
Prelude> length [1,2,3]
3
Prelude> length "Hello"
5
```

Polymorphism is one of the greatest strengths of Haskell. We will see more of this in time to come.

Let us now define a simple haskell function.

```
fac 0 = 1
fac n = n * fac (n - 1)
```

Save this in a file, say `fac.hs` and load it in the interpreter

```
$ ghci fac.hs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Main             ( fac.hs, interpreted )
Ok, modules loaded: Main.
Main> fac 0
1
Main> fac 4
24
```

A haskell function is defined by giving a set of equations. A function equation looks like

```
f pat_1_1 pat_1_2 ... pat_1_n  = expr_1
f pat_2_1 pat_2_2 ... pat_2_n  = expr_2
          ...
f pat_m_1 pat_2_2 ... pat_m_n  = expr_m
```

The formal parameters can be *patterns*. We will define what patterns in detail latter on but a constant like `0` or a variable like `n` is a pattern. When a function is evaluated, its actual arguments are *matched* with each of the equations in turn. We cannot explain what matching means in full detail here because we have not explained patterns yet. However, for constant patterns like `0` or a variable it is easy. An actual argument can match a constant if and only if the argument evaluates to that constant. On the other hand a variable pattern matches any argument. If a function `f` is defined using n equations, then while evaluating the function on a set of arguments each equation is tried out in order. The first equation of `f` whose formal parameters match the actual argument is used for evaluation.

To illustrate we consider the function `fac` that we defined. The expression `fac 0` evaluates to 1 because the actual parameter matches with the formal parameter 0 in the first equation. On the other hand, in the expression `fac 2` the actual argument `2` cannot match the formal argument 0 in the first equation of `fac`. Therefore the next equation is tried namely `fac n = n * fac (n-1)`. Here the formal parameter `n` matches 2 (recall a variable pattern can match any argument). Therefore, `fac 2` evaluates to `2 * fac (2 - 1)` which by recursion evaluates to 2.

We give two alternate defintions of the factorial function, the first using *guards* and the next using `if then else`.

```
fac1 n | n == 0      = 1
       | otherwise   = n * fac1 (n -1)

fac2 n = if n == 0 then 1 else n * fac2 (n -1)
```

To summarise

1. Haskell functions are polymorphic
2. They can be recursive.
3. One can use pattern matching in their definition.

Haskell type	What they are
Bool	Boolean type
Int	Fixed precision integer
Char	Character
Integer	Multi-precision integer

Lecture 3

More on pattern matching.

In the last lecture we, defined the factorial function and illustrated the use of pattern matching. In this chapter we elaborate on it a bit more.

3.1 Pattern matching on lists.

We have already seen the list type. There is an algebraic way of defining a list. A list either an empty list or an element attached to the front of an already constructed list. An empty list in Haskell is expressed as `[]` where as a list whose first element is `x` and the rest is `xs` is denoted by `x:xs`. The notation `[1,2,3]` is just a *syntactic sugar* for `1:(2:(3:[]))` or just `1:2:3:[]` as `:` is a right associative operator. We now illustrate pattern matching on list by giving the example of the `map` function. The `map` function takes a function and a list and applies the function on each element of the list. Here is the complete definition including the type

```
> import Prelude hiding (map, null, curry, uncurry)
>        -- hide Prelude functions defined here
>
> map :: (a -> b) -> [a] -> [b]
> map f []          = []
> map f (x:xs)      = f x : map f xs
```

Since a list can either be empty or of the form `(x:xs)` this is a complete definition. Also notice that pattern matching of variables is done here.

3.2 Literate programming detour.

Before we proceed further, let us clarify the >'s at the beginning of the of the lines. This is the literate programming convention that Haskell supports. Literate programming is a style of programming championed by Knuth, where comments are given more importance than the code. It is not restricted to Haskell alone; in fact TeX and METAFONT were written first written by Knuth in a literate version Pascal language called WEB and later on ported to CWEB, a literate version of the C Programming language. The `ghc` compiler and the `ghci` interpreter supports both the literate and non-literate version of Haskell.

Normally any line is treated as part of the program unless the commented. In literate haskell all lines other than

1. Those that start with a '>' or
2. A block of lines enclosed in a \begin{code} \end{code}

are treated as comments. We will use this literate style of programming; we will use only the first form i.e. program lines start with a >. The advantage is that one can download the notes directly and compile and execute them.

3.3 Wild card patterns.

We now illustrate the use of wild card patterns. Consider the function that tests whether a list is empty. This can be defined as follows.

```
> null [] = True
> null _  = False
```

The pattern _ (under score) matches any expression just like a variable pattern. However, unlike a variable pattern where the matched value is bound to the variable, a wild card discards the value. This can be used when we do not care about the value in the RHS of the equation.

3.4 Tuples and pattern matching.

Besides lists Haskell supports the tuple type. Tuple types corresponds to taking set theoretic products. For example the tuple `(1,"Hello")` is an ordered pair consisting of the integer `1` and the string `"Hello"`. Its type is `(Int,String)` or equivalently `(Int,[Char])` as `String` is nothing but `[Char]`. We illustrate the pattern matching of tuples by giving the definition of the standard functions `curry` and `uncurry`.

3.5 A brief detour on currying

In haskell functions are univariate functions unlike other languages. Multi-parameter functions are captured using the process called *currying*. A function taking two arguments `a` and `b` and returning `c` can be seen as a function taking the `a` and returning a function that takes `b` and returning `c`. This kind of function is called a `curried` function. Another way in which we can represent a function taking 2 arguments is to think of the function as taking a tuple. This is its uncurried form. We now define the higher order functions that transforms between these two forms.

```
> curry    :: ((a,b) -> c)  -> a -> b -> c
> uncurry  :: (a -> b -> c) -> (a,b) -> c
>
> curry   f a b   = f (a,b)
> uncurry f (a,b) = f a b
```

The above code clearly illustrates the power of Haskell when it comes to manipulating functions. Use of higher order functions is one of the features that we will find quite a bit of use.

3.6 Summary

In this lecture we saw

1. Pattern matching for lists,
2. Tuples and pattern matching on them,
3. Literate haskell
4. Higher order functions.

Lecture 4

The Sieve of Eratosthenes.

In this lecture, we look at an algorithm to find primes which you might have learned in school. It is called the Sieve of Eratosthenes. The basic idea is the following.

1. Enumerate all positive integers starting from 2.
2. Forever do the following
3. Take the smallest of the remaining uncrossed integer say p and circle it.
4. Cross out all numbers that are the factors of the circled integer p.

All the circled integers are the list of primes. For an animated description see the wiki link `http://en.wikipedia.org/Sieve_of_Eratosthenes`.

We want to convert this algorithm to haskell code. Notices that the Sieve seperates the primes from the composites. The primes are those that are circled and composites are those that are crossed. So let us start by defining

```
> primes = circledIntegers
```

i.e. the primes are precisely the list of circled integers.

An integer gets circled if and only if it is not crossed at any stage of the sieving process. Furthermore, a integer gets crossed in the stage when its least prime factor is circled. So to check whether an integer is crossed all we need to do is to check whether there is a prime which divides it.

```
> divides x n = n `mod` x == 0
>
> check (x:xs) n | x * x > n      = True       -- the rest are bigger.
>                | x `divides` n  = False      -- We hit a factor
>                | otherwise      = check xs n -- need to do more work
>
> isCircled = check primes
>
```

The function `isCircle x` checks whether `x` will eventually be circled. One need not go over all primes as the smallest non-trivial prime p that divides a composite number n should always be less than or equal to $\sqrt{n}$. This explains the first guard statement of the check function.

4.1 Guards detour.

We now explain another feature of Haskell namely guards. Look at the definition of the function `check`. Recall that a function is defined by giving a set of equations. For each such equation, we can have a set of guards. The syntax of these guarded equation looks like

```
f p_1 ... p_n | g_1  = e_1
              | g_2  = e_2
              | g_3  = e_3
              | ...
              | g_m  = e_m
```

Each of the guards `g_i` is a boolean expression. You should read this as "if `f`'s arguments match the patterns `p1 ... pn` then its value is `e_1` if `g_1` is true, `e_2` if `g_2` is true, etc `e_m` if `g_m` is true". If multiple guards are true then the guard listed first has priority. For example consider the following function

```
>
> f x | x >= 0 = "non-negative"
>     | x <= 0 = "negative"
```

Then `f 0` is the string `"non-negative"`. If you want to add a default guard, then use the keyword `otherwise`. The keyword `otherwise` is nothing but the boolean value `True`. However, in guards it is a convention to write `otherwise` instead of `True`.

Finally, we want to define the list of `circledInteger`. Clearly the first integer to be circled is 2. The rest are those integers on which the function `isCircled` returns true. And here is the Haskell code.

```
>
> circledIntegers = 2 : filter isCircled [3..]
>
```

Here filter is a function that does what you expect it to do. Its first argument is a predicate, i.e it take an element and returns a boolean, and its second argument is a list. It returns all those elements in the list that satisfies the predicate. The type of filter is given by `filter :: (a -> Bool) -> [a] -> [a]`. This completes the program for sieving primes. Now load the program into the interperter

```
$ ghci Sieve-of-Eratosthense.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Main             ( src/lectures/Sieve-of-Eratosthenes.lhs, interpreted )
Ok, modules loaded: Main.
*Main> :type take
take :: Int -> [a] -> [a]
*Main> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The `take n xs` returns the first `n` elements of the list `xs`.

4.2 How is the circularity of primes handled?

One thing you might have noticed is that the list `primes` has a circular definition. The compiler is able to grok this circularity due to the fact that Haskell is a lazy language. No expression is evaluated unless it is required. For each integer in the list `primes` to decide that it is circled, we need to consider only the primes that are less than its square root. As a result the definition of primes does not blow up into an infinitary computation.

4.3 Infix application of function and prefix application of operators.

We now explain another feature of a Haskell. Notice the use of `mod` in the expression n ‘mod‘ x == 0 in the definition of the function `divides` and in the guard x ‘divides‘ n in the definition of the function `check`. We have used a two argument function (i.e. its type is `a -> b -> c`) like an operator. For any such function `foo` we can convert it into a binary operator by back quoting it, i.e. ‘foo‘.

The converse is also possible, i.e. we can convert an operator into the corresponding function by enclosing it in a bracket. For example the expression `(+)` (there is no space in side the bracket otherwise it is an error) is a function `Int -> Int -> Int`.

```
> incr1 = map ((+) 1)
> incr2 = map (+1)
```

The two function `incr1` and `incr2` both does the same thing; increments all the elements in a list of integers by 1.

Lecture 5

Fibonacci Series

In continuation with the theme of the last lecture we define another infinite series — the fibonacci series. Ofcourse all of you know the Fibonacci Series. It is defined by the linear recurrence relation $F_{i+2} = F_i + F_{i+1}$. We assume that $F_0 = 1$ and $F_1 = 1$ to begin with.

The defintion is straight forward; it is just a one liner, but we use this as an excuse to introduce two standard list functions

5.1 Ziping a list

The zip of the list $x_0, ..., x_n$ and $y_0, \ldots, y_m$ is the list of tuples $(x_0, y_0), \ldots, (x_k, y_k)$ where k is the minimum of n and m.

```
> zip :: [a] -> [b] -> [(a,b)]
> zip  []       _    = []
> zip   _      []    = []
> zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

The function `zipWith` is a general form of ziping which instead of tupling combines the values with an input functions.

```
> zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
> zipWith f xs ys = map (uncurry f) $ zip xs ys
```

We can now give the code for fibonacci numbers.

```
> fib = 1:1:zipWith (+) fib (tail fib)
```

Notice that the zip of `fib` and `tail fib` gives a set of tuples whose caluse are consecutive fibonacci numbers. Therefore, once the intial values are set all that is required is zipping through with a `(+)` operation.

Lecture 6

Folding Lists

Will be up soon.

Lecture 7

Data types

We have already seen an example of a compound data type namely list. Recall that, a list is either an empty list or a list with a head element and rest of the list. We begin by defining a list data type. Haskell already provides a list data type so we do not need to define a user defined data type. However, we do this for illustration

```
> import Prelude hiding (sum) -- hide the standard sum function
> data List a = EmptyList
>             | Cons  a (List a)
```

One reads this as follows "List of `a` is either `EmptyList` or a `Cons` of `a` and `List` of `a`". Here the variable `a` is a type variable. The result of this is that `List` is now a polymorphic data type. We can instatiate this with any other Haskell data types. A list of integers is then `List Integer`.

The identifiers `EmptyList` and `Cons` are the two *constructors* of our new data type `List`. The constructors can now be used in Haskell expressions. For example `EmptyList` is a valid Haskell expression. So is `Cons 2 EmptyList` and `Cons 1 (Cons 2 EmptyList)`. The standard list actually has two constructors, namely `[]` and `(:)`.

7.1 Pattern Matching

We can now define functions on our new data type `List` using pattern matching in the most obvious way. Here is a version of `sum` that works with `List Int` instead of `[Int]`

```
> sum :: List Int -> Int
> sum EmptyList   = 0
> sum (Cons x xs) = x + sum xs
```

As the next example, we two functions to convert from our list type to the standard list type.

```
> toStdList   :: List a -> [a]
> fromStdList :: [a] -> List a

> toStdList EmptyList   = []
> toStdList (Cons x xs) = x : toStdList xs

> fromStdList []     = EmptyList
> fromStdList (x:xs) = Cons x (fromStdList xs)
```

1. **Exercise**: Define the functions `map` `foldr` and `foldl` for our new list type.

7.2 Syntax of a data type

We now give the general syntax for defining data types.

```
data Typename tv_1 tv_2 tv_n = C1 te_11 te_12 ... te_1r1
                             | C2 te_21 te_22 ... te_2r2
                             |      . . .
                             | Cm te_m1 te_m2 ... te_mrm
```

Here data is a key word that tells the complier that the next equation is a data type definition. This gives a polymorphic data type with *n type arguments* `tv_1,...,tv_n`. The `te_ij`'s are arbitrary type expressions and the identifiers `C1` to `Cm` are the constructors of the type. Recall that in Haskell there is a constraint that each variable, or for that matter type variable, *should* be an identifer which starts with a lower case alphabet. In the case of type names and constructors, they *should* start with upper case alphabet.

7.3 Constructors

Constructors of a data type play a dual role. In expressions they behave like functions. For example in the `List` data type that we defined the `EmptyList` constructor is a constant List (which is the same as 0-argument function) and `Cons` has type `a -> List a -> List a`. On the other hand constructors can be used in pattern matching when defining functions.

7.4 The Binary tree

We now look at another example the binary tree. Recall that a binary tree is either an empty tree or has root and two children. In haskell this can be captured as follows

```
>
> data Tree a = EmptyTree
>             | Node (Tree a) a (Tree a)
>
```

To illustrate function on tree let us define the `depth` function

```
> depth :: Tree a -> Int
> depth EmptyTree            = 0
> depth (Node left _ right) | dLeft <= dRight = dRight + 1
>                           | otherwise       = dLeft  + 1
>       where dLeft  = depth left
>             dRight = depth right
```

Lecture 8

An expression evaluator

Our goal is to build a simple calculator. We will not worry about the parsing as that requires input output for which we are not ready yet. We will only build the expression evaluator. We start by defining a data type that captures the syntax of expressions. Our expressions are simple and do not have variables.

```
> data Expr = Const Double
>           | Plus  Expr Expr
>           | Minus Expr Expr
>           | Mul   Expr Expr
>           | Div   Expr Expr
```

Now the evaluator is easy.

```
> eval :: Expr -> Double
> eval (Const d)     = d
> eval (Plus  a b)  = eval a + eval b
> eval (Minus a b)  = eval a - eval b
> eval (Mul   a b)  = eval a * eval b
> eval (Div   a b)  = eval a / eval b
```

8.1 The `Either` data type

You might have noticed that that we do not handle the division by zero errors So we want to capture functions that evaluates to a value or returns an error.

The Haskell library exports a data type called `Either` which is useful in such a situation. For completeness, we give the definition of `Either`. You can use the standard type instead.

```
data Either a b = Left  a
                | Right b
```

The data type `Either` is used in haskell often is situation where a computation can go wrong like for example expression evaluation. It has two constructors `Left` and `Right`. By convention `Right` is used for the correct value and Left for the error message: A way to remeber this convention is to remember that "Right is always right".

We would want our evaluator to return either an error message or a double. For convenience, we capture this type as `Result`.

```
> type Result a = Either String a
```

The above expression is a *type alias*. This means that `Result a` is the same type as `Either String a`. As far as the compiler is concerned, they are both same. We have already seen a type aliasing in action namely `String` and `[Char]`.

We are now ready to define the new evaluator. We first give its type

```
> eval' :: Expr -> Result Double
```

The definition of `eval'` is similar to eval for constructors `Const`, `Plus`, `Minus` and `Mul` except that it has to ensure.

1. Ensure each of the sub expressions have not resulted a division by zero error.
2. If the previous condition is met has to wrap the result into a `Result` data type using the `Right` constructor.

Instead of explicit pattern matching we define a helper function for this calle `app` that simplify this.

```
> zeroDivision = Left "Division by zero"

> app op (Right a)  (Right b) = Right (op a b)
> app _      _          _     = zeroDivision
```

The constructor Div has to be handled seperately as it is the only operator that generates an error. Again we write a helper here.

```
> divOp (Right a) (Right b) | b == 0     = zeroDivision
>                           | otherwise  = Right (a / b)
> divOp    _         _                   = zeroDivision -- problem any way.
```

Now we are ready to define eval'.

```
> eval' (Const d)   = Right d
> eval' (Plus  a b) = app (+) (eval' a) (eval' b)
> eval' (Minus a b) = app (-) (eval' a) (eval' b)
> eval' (Mul   a b) = app (*) (eval' a) (eval' b)
> eval' (Div   a b) = divOp   (eval' a) (eval' b)
```

Let us load the code in the interpreter

```
$ ghci src/lectures/An-expression-evaluator.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Main             ( src/lectures/Expression.lhs, interpreted )
Ok, modules loaded: Main.
*Main> let [zero,one,two,three,four,five] = map Const [0..5]
*Main> eval' (Div five zero)
Left "Division by zero"
*Main> eval' (Plus (Div five zero) two)
Left "Division by zero"
*Main> eval' (Plus (Div five two) two)
Right 4.5
*Main>
```

Lecture 9

Lambda Calculus

We now look at lambda calculus, the theoretical stuff that underlies functional programming. It was introduced by Alonzo Church to formalise two key concepts when dealing with functions in mathematics and logic namely: function definition and function application. In this lecture, we build enough stuff to get a lambda calculus evaluator.

```
> module Lambda where
> import Data.List -- I need some standard list functions
```

We start with an example. Consider the squaring function, i.e. the function that maps x to x^2. In the notation of lambda calculus this is denoted as $\lambda x.x^2$. This is called a lambda abstraction. Apply a function f on an expression N is written as fN. The key rule in expression evaluation is the β-reduction: the expression $(\lambda x.M)N$ reduces under β-reduction to the expression M with N substituted in it. We now look at lambda calculus formally.

The goal of this lecture is to introduce basics of lambda calculus and on the way implement a small lambda calculus interpreter.

9.1 Abstract syntax

As with any other formal system we first define its abstract syntax. A lambda calculus expression is defined via the grammar

$$e := v|e_1e_2|\lambda x.e$$

Here e_1 and e_2 expressions themselves. We now capture this abstract syntax as a Haskell data type

```
> -- | The datatype that captures the lambda calculus expressions.
> data Expr = V    String       -- ^ A variable
>           | A    Expr   Expr  -- ^ functional application
>           | L    String Expr  -- ^ lambda abstraction
>           deriving Show
```

9.2 Free and bound variables

The notion of free and bound variables are fundamental to whole of mathematics. For example in the integral $\int \sin xydy$, the variable x occurs *free* where as the variables y occurs *bound* (to the corresponding $\int dy$). Clearly the value of the expression does *not* depend on the bound variable; in fact we can write the same integral as $\int \sin xtdt$.

In lambda calculus we say a variable occurs bound if it can be linked to a lambda abstraction. For example in the expression $\lambda x.xy$ the variable x is bound where as y occurs free. A variable can occur free as well as bound in an expression — consider x in $\lambda y.x(\lambda x.x)$.

Formally we can define the free variables of a lambda expression as follows.

$$FV(v) = \{v\}$$
$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$
$$FV(\lambda x.e) = FV(e) \setminus \{x\}$$

We turn this into haskell code

```
> freeVar :: Expr -> [String]
> freeVar (V x  ) = [x]
> freeVar (A f e) = freeVar f ‘union‘ freeVar e
> freeVar (L x e) = delete x $ freeVar e
```

9.3 Variable substitution

We often need to substitute variables with other expressions. Since it is so frequent we give a notation for this. By $M[x := e]$, we mean the expression obtained by replacing all free occurrence of x in M by e. Let us code this up in haskell.

```
> subst :: Expr -> String -> Expr -> Expr
> subst var@(V y)   x e | y == x    = e
>                       | otherwise = var
> subst (A f a) x e                 = A (subst f x e) (subst a x e)
> subst lam@(L y a) x e | y == x    = lam
>                       | otherwise = L y (subst a x e)
```

9.4 Change of bound variables (α-reduction)

You are already familiar with this in mathematics. If we have an integral of the kind $\int xtdt$ we can rewrite it as $\int xydy$ by a change of variable. The same is true for lambda calculus. We say call the "reduction" $\lambda x.M \leftarrow \lambda t.M[x := t]$ as the α-reduction. However care should be taken when the new variable is chosen. Two pitfalls to avoid when performing α-reduction of the expression $\lambda x.M$ to $\lambda t.M[x := t]$ is

1. The variable t should not be free in M for otherwise by changing from x to t we have bound an otherwise free variable. For example if $M = t$ then $\lambda t.M[x = t]$ becomes $\lambda t.t$ which is clearly wrong.
2. If M has a free occurrence of x in the scope of a bound occurrence of t then we cannot perform change the bound variable x to t. For example consider $M = \lambda t.xt$. Then $\lambda t.M[x = t]$ will become $\lambda t.\lambda t.tt$ which is clearly wrong.

Clearly, one safe way to do α-reduction on $\lambda x.M$ is to use a fresh variable t, i.e. a variable that is neither free nor bound in M. We write a helper function to compute all the variables of a lambda calculus expression.

```
> varsOf :: Expr -> [String]
> varsOf (V x)   = [x]
> varsOf (A f e) = varsOf f `union` varsOf e
> varsOf (L x e) = varsOf e `union` [x]
```

We now give the code to perform a safe change of bound variables.

```
> alpha :: Expr -> [String] -> Expr
> alpha (A f e) vars                  = A (alpha f vars) (alpha e vars)
> alpha (L x e) vars | x `elem` vars  = L t $ alpha e' vars
>                    | otherwise      = L x $ alpha e  vars
>       where t  = fresh (varsOf e `union` vars)
>             e' = subst e x (V t)
> alpha  e       _                        = e
```

9.5 Function evaluation (β-reduction)

The way lambda calculus captures computation is through β reduction. We already saw what is β reduction. Under beta reduction, an expression $(\lambda x.M)N$ reduces to $M[x := N]$, where $M[x := N]$ denotes substitution of *free* occurrences of x by N. However, there is a chance that a free variable of N could become bound suddenly. For example consider N to be just y and M to be $\lambda y.xy$. Then reducing $(\lambda x.M)N$ to $M[x := N]$ will bind the free variable y in N.

We now give the code for β reduction. It performs one step of beta reduction that too if and only if the expression is of the form $(\lambda x.M)N$.

```
> beta :: Expr -> Expr
> beta (A (L x m) n) = carefulSubst m x n
> carefulSubst m x n = subst (alpha m $ freeVar n) x n
>
```

9.6 Generating Fresh variables

We saw that for our evaluator we needed a source of fresh variables. Our function fresh is given a set of variables and its task is to compute a variable name that does not belong to the list. We use diagonalisation, a trick first used by Cantor to prove that Real numbers are of strictly higher cardinality than integers.

```
> fresh :: [String] -> String
> fresh = foldl diagonalise "a"
>
> diagonalise  []        []              = "a" -- diagonalised any way
> diagonalise  []      (y:ys) | y == 'a'  = "b" -- diagonalise on this character
>                             | otherwise = "a"
> diagonalise  s        []              =  s  -- anyway s is differnt from t
> diagonalise s@(x:xs) (y:ys) | x /= y    = s   -- differs at this position anyway
>                             | otherwise = x : diagonalise xs ys
>
```

9.7 Exercise

1. Read up about β-normal forms. Write a function that converts a lambda calculus expression to its normal form if it exists. There are different evaluation strategies to get to β-normal form. Code them all up.
2. The use of `varOf` in α-reduction is an overkill. See if it can be improved.
3. Read about η-reduction and write code for it.

Lecture 10

Modules

Will be up soon.

Lecture 11

Towards type inference

A powerful feature of Haskell is the automatic type inference of expressions. In the next few lectures, we will attempt to give an idea of how the type inference algorithm works. Ofcourse giving the type inference algorithm for the entire Haskell language is beyond the scope of this lecture so we take a toy example. Our aim is to give a complete type inference algorithm for an enriched version of lambda calculus, that has facilities to do integer arithmetic. Therefore, our lambda calulus expressions have, besides the other stuff we have seen, integer constants and the built in function '+'. We limit ourselves to only one operator because it is straight forward to extend our algorithm to work with other operation

11.1 Syntax of our Enriched Lambda calculus.

The syntax is given below. Here v and x stands for arbitrary variable and e_1,e_2 stands for arbitrary expressions.

$$e = ...| -1|0|1|...| + |v|e_1 e_2|\lambda x.e$$

We will write the lambda calculus expression $+e1e2$ in its infix form $e1 + e2$ for ease of readability.

The haskell datatype that captures our enriched lambda calculus expression is the following

```
> module Lambda where
>
> -- | The enriched lambda calculus expression.
> data Expr = C Integer      -- ^ A constant
>           | P              -- ^ The plus operator
>           | V String       -- ^ The variable
>           | A Expr Expr    -- ^ function application
>           | L String Expr  -- ^ lambda abstraction
>       deriving (Show, Eq, Ord)
```

Clearly stuff like `A (C 2) (C 3)` are invalid expressions but we ignore this for time being. One thing that the type checker can do for us is to catch such stuff.

11.2 Types

We now want to assign types to the enriched lambda calculus that we have. As far as we are concerned the types for us are

$$t = \mathbf{Z} | \alpha | t_1 \to t_2$$

Here α is an arbitrary type variable. Again, we capture it in a Haskell datatype.

```
> data Type = INTEGER
>           | TV String
>           | TA Type Type deriving (Show, Eq, Ord)
```

11.3 Conventions

We will follow the following convention when dealing with type inference. The lambda calculus expressions will be denoted by Latin letters e, f, g etc with appropriate subscripts. We will reserve the Latin letters x, y, z and t for lambda calculus variables. Types will be represented by the Greek letter τ and σ with the letters α and β reserved for type variables.

11.4 Type specialisation

The notion of type specialisation is intuitivly clear. The type $\alpha \to \beta$ is a more general type than $\alpha \to \alpha$. We use $\sigma \leq \tau$ to denote the fact that σ is specialisation of τ. How do we formalise this notion of specialisation ? Firstly note that any constant type like for example integer cannot be specialised further. Secondly notice that a variable α can be specialised to a type τ as long as τ does not have an occurance of α in it. We will denote a variable specialisation by $\alpha \leftarrow \tau$.

When we have a set of variable specialisation we have to ensure that there is no cyclicity indirectly. We doe this as follows. We say a sequence $\Sigma = \{\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n\}$ is a consistent set of specialisation if for each i, τ_i does not contain any of the variables α_j, $1 \leq j \leq i$. Now we can define what a specialisation is. Given a consistent sequence of specialisation Σ let $\tau[\Sigma]$ denote the type obtained by substituting for variables in τ with their specialisations in Σ. Then we say that $\sigma \leq \tau$ if there is a specialisation Σ such that $\tau[\Sigma] = \sigma$. The specialisation order gives a way to compare two types. It is not a partial order but can be converted to one by appropriate quotienting. We say two types τ σ are isomorphic, denoted by $\sigma \equiv \tau$ if $\sigma \leq \tau$ and $\tau \leq \sigma$. It can be shown that $\equiv$ forms an equivalence relation on types. Let $\lceil \tau \rceil$ denote the equivalence class associated with τ then, it can be show that $\leq$ is a partial order on $\lceil \tau \rceil$.

11.5 Type environment

Recall that the value of a *closed* lambda calculus expression, i.e. a lambda calculus expression with no free variables, is completely determined. More generally, given an expression M, its value depends only on the free variables in it. Similary the type of an expression M is completely specified once all its free variables are assigned types. A *type environment* is an assignment of types to variables. So the general task is to infer the type of a lambda calculus expression M in a given type environment Γ where all the free varaibles of M have been assigned types. We will denote the type environments with with capital Greek letter Γ with appropriate subscripts if required. Some notations that we use is the following.

1. We write $x :: \tau$ to denote that the variable x has been assigned the type τ.
2. For a variable x, we use $\Gamma(x)$ to denote the type that the type environment Γ assigns to x.
3. We write $x \in \Gamma$ if Γ assigned a type for the variable x.
4. The type environment $\Gamma_1 \cup \Gamma_2$ denotes the the type environment Γ such that $\Gamma(x) = \Gamma_2(x)$ if $x \in \Gamma_2$ and $\Gamma_1(x)$ otherwise, i.e. the second type environment has a precedence.

As we described before, given a type environment Γ, the types of any lambda calculus expression whose free variables are assigned types in Γ can be infered. We use the notation $\Gamma \vdash e :: \tau$ to say that under the type environment Γ one can infer the type τ for e.

The type inference is like theorem proving: Think of infering $e :: \tau$ as proving that the expression e has type τ. Such an inference requires a set of rules which for us will be the type inference rules. We express this inference rules in the following notation

$$\frac{\text{Premise 1}, \ldots, \text{Premise n}}{\text{conclusion}}$$

The type inference rules that we have are the following

Rule *Const* :

$$\frac{}{\Gamma \vdash n :: \mathbf{Z}}$$

where n is an arbitrary integer.

Rule *Plus* :

$$\frac{}{\Gamma \vdash + :: \mathbf{Z} \to \mathbf{Z} \to \mathbf{Z}}$$

Rule *Var* :

$$\frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau}$$

Rule *Apply* :

$$\frac{\Gamma \vdash f :: \sigma \to \tau, \quad \Gamma \vdash e :: \sigma}{\Gamma \vdash fe :: \tau}$$

Rule *Lambda* :

$$\frac{\Gamma \cup \{x :: \sigma\} \vdash e :: \tau}{\Gamma \vdash \lambda x.e :: \sigma \to \tau}$$

Rule *Specialise* :

$$\frac{\Gamma \vdash e :: \tau, \sigma \leq \tau}{\Gamma \vdash e :: \sigma}$$

The goal of the type inference algorithm is to infer the most general type, i.e. Given an type environment Γ and an expression e find the type τ that satisfies the following two conditions

1. $\Gamma \vdash e :: \tau$ and,
2. If $\Gamma \vdash e :: \sigma$ then $\sigma \leq \tau$.

11.6 Exercises

1. A pre-order is a relation that is both reflexive and transitive.
    - Show that the specialisation order $\leq$ defined on types is a pre-order.

- Given any pre-oder $\preceq$ define the associated relation $\simeq$ as $a \simeq b$ if $a \preceq b$ and $b \preceq a$. Prove that $\simeq$ is an equivalence class. Show that $\preceq$ can be converted into a natural partial order on the equivalence class of $\simeq$.

2. Prove that if σ and τ are two types such that $\sigma \equiv \tau$ then prove that there is a bijection between the set $Var(\sigma)$ and $Var(\tau)$ given by $\alpha_i \mapsto \beta_i$ such that $\sigma[\Sigma] = \tau$ where Σ is a specialisation $\{\alpha_i \leftarrow \beta_i | 1 \leq i \leq n\}$. In particular isomorphic types have same number of variables. (Hint: use induction on the number of variables that occur in σ and τ).

Lecture 12

Unification algorithm

On of the key steps involved in type inference is the unification algorithm. Given two type τ_1 and τ_2 a *unifier* if it exists is a type τ such that τ is a specialisation of both τ_1 and τ_2. The *most general unifier* of τ_1 and τ_2, if it exists, is the unifier which is most general, i.e. it is the unifier τ^* of τ_1 and τ_2 such that all other unifiers τ is a specialisation of τ^*. In this lecture we develop an algorithm to compute the most general unifier of two types τ_1 and τ_2.

```
> {- | Module defining the unification algorithm -}
>
> module Unification where
> import Lambda                   -- See the last lecture
> import qualified Data.Map as M -- To define specialisation.
> import Data.Either
> import Data.Maybe
>
> type Map = M.Map
```

Although we defined the unifier as a type, it is convenient when computing the unifier of τ_1 and τ_2 to compute the type specialisation Σ that unifies them to their most general unifier. We will therefore abuse the term most general unifier to also mean this specialisation. Also for simplicity we drop the adjective "most general" and just use unifier to mean the most general unifier.

```
> type Specialisation = Map String Type -- ^ A type specialisation
> type Result         = Either String   -- ^ The error message if it failed
>                        Specialisation   -- ^ The unifier
>
> unify               :: Type -> Type -> Result
```

A type specialisation is captured by a `Map` from type variable names to the corresponding specialisation. We given an function to compute $\tau[\Sigma]$ given a specialisation Σ.

```
> specialise :: Specialisation -> Type -> Type
> specialise sp t@(TV x) = maybe t (specialise sp) $ M.lookup x sp
> specialise sp (TA s t) = TA (specialise sp s) (specialise sp t)
> specialise sp  i       = i
```

We generalise the unification in two ways:

1. We consider unification of types under a particular specialisation Σ. The unifier of τ_1 and τ_2 *under the specialisation* Σ is nothing but the unifier of $\tau_1[\Sigma]$ and $\tau_2[\Sigma]$. The unification of two types can then be seen as unification under empty specialisation.

```
> genUnify  :: Specialisation -> Type -> Type -> Result
> unify     = genUnify M.empty
```

2. Instead of considering unifiers of pairs of types τ_1 and τ_2, we consider the simultaneous unifier of a sequence of pairs $\{(\tau_1, \sigma_1), \ldots, (\tau_n, \sigma_n)\}$. A unifier of such a sequence is a specialisation Σ such that $\tau_i[\Sigma] = \sigma_i[\Sigma]$ for all $1 \leq i \leq n$. Of course we want to find the most general of such unifier. We call this function `genUnify'`. Given, `genUnify` it is easy to define `genUnify'`.

```
> genUnify'  :: Specialisation  -- ^ pair of types to unify
>            -> [(Type,Type)]   -- ^ the specialisation to unify under
>            -> Result          -- ^ the resulting unifier
> genUnify' = foldl fld . Right
>           where fld (Right sp) (tau,sigma) = genUnify sp tau sigma
>                 fld err            _       = err
```

What is left is the definition of `genUnify`.

```
> genUnify sp (TV x)   t       = unifyV sp x t
> genUnify sp t        (TV x)  = unifyV sp x t
> genUnify sp INTEGER  t       = unifyI sp t
> genUnify sp t        INTEGER = unifyI sp t
> genUnify sp ap1      ap2     = unifyA sp ap1 ap2
>
```

The order of the pattern matches are important. For example notice that in line 6, both `ap1` and `ap2` have to be an arrow type (why?) Also in lines 3 and 4, t can either be `INTEGER` or an arrow type.

So our rules of unification can are split into unifying a variable α with a type τ, a constant types (here Integer) with a type τ and unifying two arrow types. We capture these rules with functions `unifyV`, `unifyI` and `unifyA` respectively.

```
> unifyV :: Specialisation -> String -> Type -> Result
> unifyI :: Specialisation -> Type   -> Result
> unifyA :: Specialisation -> Type   -> Type -> Result
```

12.1 Unifying a variable and a type

Unification of variables can be tricky since specialisations are involved. Firstly, notice that if α occurs in a type τ, we cannot unify α with τ unless τ is α itself. We first give a function to check this.

```
> occurs :: Specialisation -> String -> Type -> Bool
> occurs sp x (TV y)   | x == y    = True
>                      | otherwise = maybe False (occurs sp x) $ M.lookup y sp
> occurs sp x (TA s t) = occurs sp x s || occurs sp x t
> occurs sp x  _       = False
```

The rules of unification of a variable α and a type τ are thus

1. If α has a specialisation σ, unify σ and tau,
2. If τ is α or can be specialised to α then we already have the specialisation.
3. If α is unspecialised then specialise it with τ provided there occurs no recursion either directly or indirectly.

```
> unifyV  sp x t = maybe (unifyV' sp x t) (genUnify sp t) $ M.lookup x sp
>
> unifyV' sp x INTEGER                 = Right $ M.insert x INTEGER sp
> unifyV' sp x var@(TV y)  | x == y    = Right $ sp
>                          | otherwise = maybe (Right $ M.insert x var sp)
>                                              (unifyV' sp x)
>                                              $ M.lookup y sp
> unifyV' sp x ap@(TA s t) | occurs sp x s = failV sp x ap
>                          | occurs sp x t = failV sp x ap
>                          | otherwise  = Right $ M.insert x ap sp
>
> failV sp x ap = Left $ unwords [ "Fail to unify", x, "with" , ppSp sp ap
>                                , ": recursion detected."
>                                ]
>
```

Notice here that `M.lookup x sp` returns `Nothing` if `x` is not specialised under the specialisation `sp`. The function $unifyV'$ unifies only an unspecialised variable with τ.

12.2 Unifying an integer and type

Notice here that the type τ is not a variable. Then it can only be Integer or an arrow type.

```
> unifyI sp INTEGER = Right sp
> unifyI sp t       = Left $ unwords [ "Failed to unify Integer with type"
>                                    , ppSp sp t
>                                    ]
```

12.3 Unifying two arrow types

```
> unifyA sp ap1@(TA a b) ap2@(TA c d) = either errmsg Right
>                                   $ genUnify' sp [(a,c),(b,d)]
>          where errmsg str = Left $ unwords [ "while unifying"
>                                            , ppSp sp ap1, "and"
>                                            , ppSp sp ap2
>                                            ] ++ "\n" ++ str
```

12.4 Helper functions.

We now document the helper functions used in the unification algorithm. First we give an algorithm to *pretty print* types. This makes our error messages more readable. Notice that this version does not put unnecessary brackets.

```
> -- | Pretty print a type
>
> pp :: Type -> String
> pp INTEGER  = "Integer"
> pp (TV x)   = x
> pp (TA s t) = bracket s ++ " -> " ++ pp t
>       where bracket t@(TA r s) = "(" ++ pp t ++ ")"
>             bracket s         = pp s
>
> -- | Pretty print a specialised type
> ppSp :: Specialisation -> Type -> String
> ppSp sp = pp . specialise sp
```

12.5 Testing this code using ghci

Since the lecture used the module of the previous lecture you need to give the following commandline arguments.

```
$ ghci src/lectures/Unification-algorithm.lhs src/lectures/Towards-type-inference.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 2] Compiling Lambda           ( src/lectures/Towards-type-inference.lhs, interpreted )
[2 of 2] Compiling Unification      ( src/lectures/Unification-algorithm.lhs, interpreted )
Ok, modules loaded: Unification, Lambda.
*Unification> let [a,b,c,d,e] = map TV $ words "a b c d e"
*Unification> unify (TA INTEGER a) b
Loading package array-0.3.0.2 ... linking ... done.
Loading package containers-0.4.0.0 ... linking ... done.
Right (fromList [("b",TA INTEGER (TV "a"))])
*Unification> unify (TA INTEGER a) (TA b (TA b c))
Right (fromList [("a",TA (TV "b") (TV "c")),("b",INTEGER)])
*Unification>
*Unification> let t = (TA a b)
*Unification> unify (TA INTEGER a) (TA c t)
Left "while unifying Integer -> a and c -> a -> b\nFail to unify a with a -> b : recursion detect
*Unification> let Left l = unify (TA INTEGER a) (TA c t)
*Unification> putStrLn l
while unifying Integer -> a and c -> a -> b
Fail to unify a with a -> b : recursion detected.
```

```
*Unification>
```

Lecture 13

The type inference algorithm

Will be up some day.

Lecture 14

Type classes

Consider the following interaction with the `ghci`

```
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> (+) 1

<interactive>:1:1:
    No instance for (Show (a0 -> a0))
      arising from a use of ‘print’
    Possible fix: add an instance declaration for (Show (a0 -> a0))
    In a stmt of an interactive GHCi command: print it
Prelude> (+) 1 2
3
Prelude>
```

Here the interpreter is able to display the value `3` but not the function `(+) 1`. The error message is instructive. It says that there is no instance of `Show (a0 -> a0)` for use with `print`. Let us see what `Show` and `print` are by using the `:info` command of the interpreter.

```
$ ghci
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :info Show
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
        -- Defined in GHC.Show
instance (Show a, Show b) => Show (Either a b)
  -- Defined in Data.Either
instance Show a => Show [a] -- Defined in GHC.Show
instance Show Ordering -- Defined in GHC.Show
instance Show a => Show (Maybe a) -- Defined in GHC.Show
instance Show Int -- Defined in GHC.Show
instance Show Char -- Defined in GHC.Show
instance Show Bool -- Defined in GHC.Show

[lots of similar lines delete]

Prelude> :info print
print :: Show a => a -> IO ()   -- Defined in System.IO
Prelude>
```

To understand what it means we need to know how the interpreter behaves. When even you enter an expression on the command line the interpreter evaluates and tries to print it. However not all haskell types are printable. The function `print` has the type `Show a => a -> IO ()` which means `print` argument type `a` is not a general type variable but is constranined to take types which are an instance of `Show`. Here `Show` is a type class.

Note that from `:info Show` it is clear that the standard types like `Int`, `Bool` etc are all instances of the class `Show`. Therefore the interpreter could print any value of those type. However, a function type is not an instance of `Show` and hence could not be printed.

Lecture 15

Monads and IO

Will be up soon

Lecture 16

State monads

Will be up soon

Lecture 17

Monad transformes

Will be up soon

Lecture 18

Functions with varible number of arguments.

Consider the printf function in C. The number of arguments it take depends on the format string that is provided to it. Can one have similar functions in Haskell ?

Let us analyse the type of `printf` in various expressions. In the expression `printf "Hello"`, `printf` should have type `String -> IO ()` where as in an expression like `printf "Hello %s" "world"`, it should have the type `String -> String -> IO()`. This chamelon like behaviour is what we need to define printf.

In this lecture we show how to hack Haskell's type class mechanism to simulate such variable argument functions. The stuff we will do is beyond Haskell 98 and hence we need to enable certain extensison. We do it via the following compiler pragmas. The very first set of lines in the source code if they are comments that start with `{-#` and end with `#-}` are treated as compiler pragmas. In this case we want to allow the extensions FlexibleInstances OverlappingInstances and InvoherentInstances. One can also give these extensions at compile time; to enable the extension Foo use the flag -XFoo at compile time.

Ofcourse it is difficult to remember all the extensions that you need for this particular code to work. The easiest way to know what extensions are required is to go ahead and use it in your file. GHC will warn you with an appropriate error message if it expects an extension.

```
> {-# LANGUAGE FlexibleInstances        #-}
> {-# LANGUAGE OverlappingInstances     #-}
> {-# LANGUAGE IncoherentInstances      #-}
```

```
>
> module Printf where
```

Recall that an expression using printf would look something like the one below.

```
printf fmt e1 e2 ... em
```

We want our type system to infer `IO ()` for this expression. The type of `printf` in such a case should be `String -> t1 -> ... -> tm -> IO ()` where `ti` is the type of `ei`. The main idea is to define a type classes say `Printf` whose instances are precisely those types that are of the form `t1 -> ... -> tm -> IO ()`. As a base case we will define an instance for `IO ()`. We will then inductively define it for other types.

The definition of the type class `Printf` and therefore `printf` will be easier if we first declare a data type for formating.

```
>
> data Format = L String -- ^ A literal string
>             | S        -- ^ %s
>             | G        -- ^ %g for instances of Show
>
>
```

We would rather work with the more convenient `[Format]` instead of the format string. Writing a function to convert from format string to `[Format]` is not too difficult.

Now for the actual definition of the `Printf` class.

```
> class Printf s where
>       printH :: IO [Format] -> s
>
```

The member function `printH` of the class `Printf` is a helper function that will lead us to definition of `printf`. Intutively it takes as argument an IO action which prints whatever arguments it has seen so far and returns the rest of the formating action required to carry out s.

We will define the `Printf` instances for each of the type `t1 -> ... tm -> IO ()` inductively. The base case is when there are no arguments.

```
> instance Printf (IO ()) where
>       printH fmtIO  = do fmt <- printLit fmtIO
>                          if null fmt then return ()
>                             else fail "Too few arguments provided"
>
```

Now the inductive instance declaration.

```
> instance (Printf s, Show a) => Printf (a -> s) where
>          printH fmtIO a = printH action
>                 where action = do rfmt <- printLit fmtIO
>                                   case rfmt of
>                                        []     -> fail "Too many argument"
>                                        (_:xs) -> do putStr $ show a; return xs
```

We give a specialised instance for string which depending on whether the formating character is %s or %g uses putStr or print.

```
>
> instance Printf s => Printf (String -> s) where
>          printH fmtIO s = printH action
>                 where action = do rfmt <- printLit fmtIO
>                                   case rfmt of
>                                        [] -> fail "Too many arguments"
>                                        (G:xs) -> do putStr $ show s; return xs
>                                        (S:xs) -> do putStr s; return xs
>
```

What is remaining is to define printLit

```
>
> printLit fmtIO = do fmt <- fmtIO
>                     let (lits,rest) = span isLit fmt
>                         in do sequence_ [putStr x | L x <- lits]
>                               return rest
> isLit (L _) = True
> isLit  _    = False
>
```

We can now define printf

```
> printf :: Printf s => String -> s
> printf = printH . format
>
> format :: String -> IO [Format]
> format ""         = return []
> format "%"        = fail "incomplete format string"
> format ('%':x:xs) = do fmts <- format xs
>                        case x of
>                             's' -> return (S:fmts)
>                             'g' -> return (G:fmts)
>                             '%' -> return (L "%" :fmts)
>                             _   -> fail ("bad formating character %" ++ [x])
> format zs = let (lit,xs) = span (/='%') zs
>             in do fmts <- format xs
>                   return (L lit:fmts)
>
```

Lecture 19

Template Haskell based implementation of `printf`

In the last lecture we saw a type class based solution to create functions that take variable number of arguments. Here we give a template haskell based solution.

19.1 What is Template Haskell ?

Template haskell is the Haskell way of doing Meta programming. At the very least one can use it like a macro substitution but it can be used to do much more. The idea is to process the Haskell code at compile time using Haskell itself. A programmer can write Haskell functions to manipulate Haskell code and using special syntax arrange the compiler to manipulate code at compile time. In this lecture we will see how to define a version of printf using template haskell.

Template Haskell consists of two important steps.

1. Quoting: To allow user defined function to manipulate the Haskell code one needs to represent the program as a value in some suitable data type. The data types defined in the module `Language.Haskell.TH.Syntax` is used for this purpose. For example the type `Exp` defined in the above module is used to represent a valid Haskell expression. Have a look into the documentation of that module.

2. Splicing. Once the Haskell language fragment is processed using various function defined by the user, it needs to be compiled by the compiler. This processing is called splicing.

One point to be noted though is that template haskell does not splice the code directly but only those that are expressions that are inside the quoting monad Q. This monad is required because while generate code various side effects are created. For example a variable `x` used in a fragment of the code has a different meaning if there is a local binding defined on it. Besides one would want to read in data from files (think of config files) to perform compile time operations.

There are two syantactic extensions to Haskell that makes template Haskell possible. If a haskell expression is written between `[|` and `|]`, the compiler will replace it with the corresponding representation in `Language.Haskell.TH.Syntax`. For example, the expression `[| "Hello" |]` is of type `Q Exp`. The corresponding `Exp` value is `LitE (StringL "Hello")`.

The following are the extra syntactic conventions used.

1. `[e| ... |]` or just `[| ...` '|]' for quoting expressions. This has type `Q Exp`.
2. `[d| ... |]` for quoting declarations. This has type `Q [Decl]`
3. `[t| ... |]` for quoting types. This has type `Q Type`.
4. `[p| ... |]` for quoting patterns. This has type `Q Pat`.

The splicing is done using the syntax `$(...)` (no space between `$` and `()`

19.2 Some convenience types and combinators

The types `Q Exp` and `Q Pat` etc occur so often that there are aliases for them `ExpQ` and `PatQ` respectively. As an exercise guess the types aliases for `Q Type` and `Q Decl`.

Whenever possible it is better to make use of the `[| ... |]` notation to build quoted expressions. However sometimes it is better to use the constructors of `Exp` directly. Recall that we can splice only quoted expressions, i.e values of type Q Expr (or equivalently ExpQ). Say you have a quoted expressions `qe` and `qf` which correspondes to the haskell expression `e` and `f` respectively. If one wants to obtaine the quoted expression which correspondes to the application of the function `f` on the expression `e`, we would have to do something like the following

```
qfe = do e <- qe
         f <- qf
     return $ AppE f e.
```

To make this efficient there is a combinator called `appE` which does essentially what the constructor `AppE` does but works on `Q Exp` rather than `Exp`.

```
appE :: ExpQ -> ExpQ -> ExpQ
```

The above code will then look like `appE qe qf`. There are such monadic version of all the constructors of `Exp` available. Make use of it.

19.3 Printf

First note that we have enabled template Haskell using the compiler pragma given above. It should be the very first line of your source code. A unrecognised pragma is ignored by the compiler but sometimes a warning is issued.

```
> {-# LANGUAGE TemplateHaskell    #-}
> module Printf where
> import Data.List
> import Language.Haskell.TH
```

Few things about enabling the template haskell. Strictly speaking this module does not need TemplateHaskell, rather it can be written without using template Haskell. This is because all it does is define functions that process objects of type `Expr` or `ExprQ`. I have enabled it so as to write `appE [|show|]` instead of the more complicated. `appE (varE 'show)`

First let us capture the formating via a data type

```
> data Format = L String    -- ^ literal string
>             | S           -- ^ %s
>             | G           -- ^ %g generic type
>             deriving Show
```

We need a function that will parse a string and the give the corresponding list for format. The exact details are not really of interest as far as template haskell is concerned. See the end of the file for an implementation.

```
> format :: String -> [Format]
```

The printf function can then be defined as.

```
> printf  :: String -> ExpQ
> printfP :: [Format] -> ExpQ
> printf = printfP . format
```

We would rather implement the prinfP function. Let the list of formatting instructions be [f_1,..,f_n], then we want `prinfP [f_1,...,f_n]` when spliced to return the code.

```
\ x0  ... xm -> concat [e1,...,e_n]
```

Here e_i depends on the ith format instruction f_i. If f_i is a literal then it will just be a literal string. Otherwise it would be an appropriate variable. In our case it should be `xj` where `j` is the number of *non-literal*, i.e. `S` or `G`, formating instructions to the left of e_i. The number m is the total number of *non-literal* formatting instructions in the list [f_1,. . . ,f_n] and should be less than or equal to n.

Suppose we know the number of variables to the left of a format instruction f_i, how do we generate the expression e_i ? The following function does that

```
> toExpQ :: Int         -- ^ formatting instructions to the left
>        -> Format      -- ^ The current spec.
>        -> (Int,ExpQ)  -- ^ The total number of non-literal instruction
>              -- to the left and the resulting expression.
> toExpQ i   (L s) = (i,string s)
> toExpQ i   S     = (i+1,varExp i)
> toExpQ i   G     = (i+1,showE $ varExp i)
```

Here we make use of the following helper Template haskell functions which we have defined subsequently.

```
> string  :: String -> ExpQ  -- ^ quote the string
> showE   :: ExpQ   -> ExpQ  -- ^ quoted showing
> varExp  :: Int    -> ExpQ  -- ^ quoted variable xi
> varPat  :: Int    -> PatQ  -- ^ quoted pattern xi
```

The printfP function then is simple. Recall that when spliced it should generate the expression

```
       \ x0 ... xm -> concat [e1,...,e_n]‘
```

The complete definition is given below.

```
> printfP fmts = lamE args . appE conc $ listE es
>         where (nvars,es) = mapAccumL toExpQ 0 fmts
>               args       = map varPat [0 .. (nvars-1)]
>               conc       = [|concat|]
```

Here are the definition of the helper functions

```
> string   = litE . StringL
> varExp i = varE $ mkName ("x" ++ show i)
> varPat i = varP $ mkName ("x" ++ show i)
> showE    = appE [|show|]
```

We now come to the parsing of the format string. For simplicity of implementation if % occurs before an unknow format string it is treated as literal occurance.

```
> format ""             = []
> format ['%']          = [L "%"]
> format ('%':x:xs)
>        | x == 's'  = S : format xs
>        | x == 'g'  = G : format xs
>        | x == '%'  = L "%" : format xs
>        | otherwise = L ['%',x] : format xs
> format zs          = L x : format xs
>        where (x,xs) = span (/='%') zs
```

To try it out load it with ghci using the option -XTemplateHaskell. You need this option to tell ghci that it better expect template haskell stuff like Oxford bracket [| |] and splicing `$(...)`

```
$ ghci -XTemplateHaskell src/lectures/Template-Haskell-based-printf.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Printf           ( src/lectures/Template-Haskell-based-printf.lhs, interpreted
Ok, modules loaded: Printf.
```

```
*Printf> $(printf "%s is a string") "Hello"
"Hello is a string"
*Printf> $(printf "%s is a string %g is an int") "Hello" 10
"Hello is a string 10 is an int"
*Printf>
```

19.4 Exercise

1. Write a function that will optimise the format instructions by merging consecutive literal strings. Is there any point in optimising the format instructions?

2. Rewrite the printf module to not use any template haskell itself.

Lecture 20

Concurrent programming in Haskell

Haskell has a very good support for concurrent programming. In this lecture we see a very basic introduction to concurrent programming.

20.1 Threads

The Haskell runtime implements threads which are really lightweight. This means that on a decent machine you can open say 10K threads and still be have decent performance. This makes Haskell a great platform to implement high connectivity servers like http/ftp servers etc.

However you need to be carefull about FFI calls to C code. If one of your thread makes a call to a C function that blocks then all the threads will block. Sometimes the call to C functions might be indirect, you might use a library that uses Haskells FFI to talk to an already implemented C library. Then you can be in trouble if you are not careful.

One creates threads in Haskell using the `forkIO :: IO () -> IO ThreadId` function. The threads created using `forkIO` are really local to you ghc process and will not be visible in the rest of the OS. There is a similar `forkOS` function that creates an OS level thread. As long as the code uses pure haskell functions `forkIO` is all that you need.

We give here a basic function that forks a lot of worker process. The module to import is `Control.Concurrent`.

```
> import Control.Concurrent -- Concurrency related module
> import System.Environment -- For command line args in main
```

This is the worker process. In real life program this is were stuff happen.

```
> worker :: Int -> IO ()
> worker inp = do tId <- myThreadId
>                 let say x = putStrLn (show tId ++ ": " ++ x)
>                     in do say ("My input is " ++ show inp)
>                           say "Oh no I am dying."
>
```

Here is where the process is created. Note that `forkIO . worker` takes as input an integer and runs the worker action on it in a seperate thread

```
> runThreads :: [Int] -> IO ()
> runThreads = sequence_ . map (forkIO . worker)
>
```

And finally this is the main function where all the command line arguments are parsed and things done.

```
> main = do args <- getArgs
>           case args of
>                [a] -> let nthreads = read a
>                           in runThreads [1..nthreads]
>                _   -> putStrLn "Bad arguments"
>
```

20.2 Daemonic thread termination

There is a big bug in the code you just saw. All threads are terminated as soon as the main thread terminates. For real world applications one wants main thread to hang around till the others are done. We will see one way to handle this in the next lecture.

Last modified on Monday (31 October 2011 10:14:13 UTC)