PROGRAMMING IN HASKELL

Chapter 3 - Types and Classes

(Original slides by Graham Hutton)

What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

Bool

contains the two logical values:

False True

Type Errors

Applying a function to one or more arguments of the wrong type is called a type error.


```
> 1 + Fals
Error
```

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written

e :: t

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

- In GHCi, the :type command calculates the type of an expression, without evaluating it:


```
> not False
True

> :type not False
not False :: Bool
```

Basic Types

Haskell has a number of basic types, including:

Bool	- logical values
Char	
String	- strings of chars
Int	- fixed-precision integers
Integer	- arbitrary-precision integers
Float	- floating-point numbers

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```


In general:

[t] is the type of lists with elements of type t.

Note:

- The type of a list says nothing about its length:

```
[False,True]      :: [Bool]

[False,Tru
```


- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Note:

- Strings in Haskell are just lists of characters!


```
[[’a’],[’b’,’c’]] :: [[Char]]
```


is equivalent to:

```
["a","bc"] :: [String]
```

Tuple Types

A tuple is a sequence of values of different types:

```
(False,True)     :: (Bool,Bool)

(False,'a',True) :: (Bool,Ch
```


In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

Note:

- The type of a tuple encodes its size:

```
(False,True)      :: (Bool,Bool)

(False,True,'a')  :: (Bool,Bool,Char)
```


- The type of the component ricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b'])  :: (Bool,[Char])
```

Function Types

A function is a mapping from values of one type to values of another type:


```
not     :: Bool
                  -> Bool

isDigit :: Char -> Bool
```


In general:

t1 → t2 is the type of functions that map values of type t1 to values to type t2.

Note:

- The arrow → is typed at the keyboard as ->.
- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible


```
add       :: (Int,Int) → Int
add (x,y)  = x+y

zeroto    :: Int → [Int]
zeroto n   = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:


```
add'    :: I
add' x y = x+y
```


add' takes an integer x and returns a function add' x. In turn, this function takes an integer y and returns the result x+y.

Note:

- add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:


```
add  :: (In
add' :: Int → (Int → Int)
```


- Functions that take their arguments one at a time are called curried functions, celebrating the work of Haskell Curry on such functions.

- Functions with more than two arguments can be curried by returning nested functions:

```
mult      :: Int → (Int → (Int → Int))
mult x y z = x*
```


mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function.

For example:


```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow → associates to the right.

Int → Int → Int → Int

Means Int → (Int → (Int → Int)).

- As a consequence, it is then natural for function application to associate to the left.

```
mult x y z
```


Means ((mult x) y) z.

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.


```
length :: [
```


for any type a, length takes a list of values of type a and returns an integer.

Note:

- Type variables can be instantiated to different types in different circumstances:


```
> length
2
> length [1,2,3,4]
4
```

a = Bool

a = Int

- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]

id   :: a → a
```

What's the type of sum?

What should be the type of sum (the function that sums a list of numbers)?


```
sum :: [a] -
```


We should be able to use sum s types of numbers:

```
sum [1,2,3]     —>  6
sum [1.5,2.3]   —>  3.8
```

Overloaded Functions

A polymorphic function is called overloaded if its type contains one or more class constraints.


```
sum :: Nu
```


for any numeric type a, sum takes a list of values of type a and returns a value of type a.

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:


```
> sum [1,
6
> sum [1.1,2.2,3.3]
6.6

> sum ['a','b','c']
ERROR
```


a = Int

a = Float

Char is not a numeric type

- Haskell has a number of type classes, including:

  - **Num** - Numeric types
  - **Eq** - Equality types
  - **Ord** - O

- For example:

```
(+)  :: Num a ⇒ a → a → a
(==) :: Eq a  ⇒ a → a → Bool
(<)  :: Ord a ⇒ a → a → Bool
```

Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type;

- Within a scri e to state the type of ever ed;

- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

Exercises

(1) What are the types of the following values?


```
['a','b','
```



```
('a','b','c')
```



```
[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail,init,reverse]
```

(2) What are the types of the following functions?

```
second xs     = head (tail xs)

swap (x,y)    = (y,x)

pair x y

double x      = x*2

palindrome xs = reverse xs == xs

twice f x     = f (f x)
```


(3) Check your answers using GHCi.

PROGRAMMING IN HASKELL

Chapter 4 - Defining Functions

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.


```
abs :: Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional expressions can be nested:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
```


Note:

- In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.


```
abs n | n ≥ 0 = n
      | otherwise = -n
```

As previously, but using guarded equations.

Guarded equations can be used to make definitions involving multiple conditions easier to read:

Note:

- The catch all condition otherwise is defined in the prelude by otherwise = True.

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.


```
not      :: B
not False = True
not True  = False
```


not maps False to True, and True to False.

Functions can often be defined in many different ways using pattern matching. For example

```
(&&) :: Bool -> Bool -> Bool
(&&) = ?
```


can be defined more compactly by

```
?
```

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True  && b
False && _
```


Note:

- The underscore symbol _ is a wildcard pattern that matches any argument value.

- Patterns are matched in order. For example, the following definition always returns False:

```
_    && _    = False
True && True = True
```


- Patterns may not repeat vari example, the following definition gives an error:

```
b && b = b
_ && _ = False
```

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.

[1,2,3,4]

Means 1:(2:(3:(4:[]))).

Functions on lists can be defined using x:xs patterns.

```
head      :: [a] → a
head (x:_) = x

tail      ::
tail = ?
```


head and tail map any non-empty list to its first and remaining elements.

Note:

- x:xs patterns only match non-empty lists:

```
> head []
Error
```


- x:xs patterns must be parenthesised because application has priority over (:). For example, the following definition gives an error:

```
head x:_ = x
```

Lambda Expressions

Functions can be constructed without naming the functions by using lambda expressions.

$\lambda x \rightarrow x+x$

the nameless function that takes a number x and returns the result x+x.

Note:

- The symbol λ is the Greek letter lambda, and is typed at the keyboard as a backslash \.

- In mathemati are usually denoted using x+x.

- In Haskell, the use of the λ nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based.

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:


```
add x y = x+y
```

means

```
add = λx → (λy → x+y)
```

Lambda expressions are also useful when defining functions that return functions as results.

For example:


```
const    ::
const x _
```


is more naturally defined by

```
const  :: a → (b → a)
const x = λ_ → x
```

Lambda expressions can be used to avoid naming functions that are only referenced once.

For example:


```
odds n =
        wher
          f x = x*2 + 1
```


can be simplified to

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

For example:


```
> 1+2
3

> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:


```
> (1+) 2
3

> (+2) 1
3
```


In general, if ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called sections.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

(1+)

(1/) - reciprocation fun

(*2) - doubling function

(/2) - halving function

Exercises

(1) Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

(a) a conditional expression;
(b) guarded equ
(c) pattern matc

Hint: the library function null :: [a] -> Bool is used to test if a list is empty.

(2) Give three possible definitions for the logical or operator (||) using pattern matching.

(3) Redefine the following version of (&&) using conditionals rather than patterns:


```
True &&
_    && _    = False
```


(4) Do the same for the following version:

```
True  && b = b
False && _ = False
```