Types

Since each expression has a type, each definition also has a type.

- **square** \( n = n \times n \)
  
  then square has type \( \text{Integer} \to \text{Integer} \)
  
  (Ok, it really is something more generic, but let me lie for once more.)

- **mynumber = square 10**
  
  then mynumber has type \( \text{Integer} \)
Specifying Types

• You can specify the type of an expression or a subexpression:

\[
10 + 12 :: \text{Integer} \quad \text{--specifies the whole expression}
\]
\[
10 + (12 :: \text{Integer}) \quad \text{--specifies the 12}
\]

• You can also specify the type of a definition. Write it on a separate line.

\[
square :: \text{Integer} \to \text{Integer}
\]
\[
square n = n * n
\]

• Generally, \(x :: T\) is pronounced as “\(x\) has type \(T\)”. You may omit such specifications. Then Haskell will compute the most “generic” types possible.
Defining Your Own Types

Let us define a colour type. It will be like an enumerated type.

```
data Colour = Red | Green | Blue
```

- The name of the new type is `Colour`.
- Its possible values are: Red, Green, Blue.

Note:

- Red, Green, Blue are called *constructors* of Colour: they produce values of the type.
- Type names and constructor names must begin with capital letters.
Writing Functions for Your Types

Let us define a function that maps Colour to integer RGB codes. Red goes to $255 \times 2^{16}$, Green goes to $255 \times 2^8$, Blue goes to 255.

```
toRGB :: Colour -> Int
toRGB c = case c of
    Red  -> 255 * 2^16
    Green -> 255 * 2^8
    Blue  -> 255
```

This is just like procedural languages. Is there a more elegant way?
Writing Functions for Your Types (cont.)

More elegant way:

```haskell
  toRGB :: Colour -> Int
  toRGB Red = 255 * 2^16
  toRGB Green = 255 * 2^8
  toRGB Blue = 255
```

Execution view:

- Computer compares the actual parameter with the formal parameters.
- Computer selects the first equation that matches.

This is called pattern matching.
How to Write a Function

Human conceptual view:

• I want Red to be mapped to $255 \times 2^{16}$.

  $\text{toRGB } \text{Red} = 255 \times 2^{16}$

• I also want Green to be mapped to $255 \times 2^{8}$.

  $\text{toRGB } \text{Green} = 255 \times 2^{8}$

• I also want Blue to be mapped to 255.

  $\text{toRGB } \text{Blue} = 255$

This is how you should write a function or read one.
More Examples of Functions

More functions written with pattern matching. Try to get used to them.

- Straightforward factorial.

  \[
  \text{factorial} :: \text{Integer} \rightarrow \text{Integer} \\
  \text{factorial} \ 1 = 1 \\
  \text{factorial} \ n = n \times \text{factorial} \ (n-1)
  \]

- Smart factorial.

  \[
  \text{smartfact} :: \text{Integer} \rightarrow \text{Integer} \\
  \text{smartfact} \ n = f \ 1 \ n \\
  \text{where} \ f \ p \ 1 = p \\
  \quad f \ p \ i = f \ (p \times i) \ (i-1)
  \]
A More Interesting Type

Let us define a shape type. A shape will be a rectangle or an ellipse.

- A rectangle will have a width and a height.
- An ellipse will have a width and a height too (lengths of the axes).

Kind of like a union type.

```haskell
data Shape = Rectangle Float Float
            | Ellipse Float Float
```

Now each constructor takes some parameters.
E.g., Rectangle takes two floating-point numbers, a width and a height. (Unfortunately the syntax only lets us write the types.)
A More Interesting Type (cont.)

Some expressions of type Shape:

Rectangle 1.0 2.0 :: Shape
Ellipse 2.0 3.0 :: Shape

If you enter them at a Haskell prompt, you’ll get an error message:

ERROR: Cannot find "show" function for:
*** Expression : Rectangle 1.0 2.0
*** Of type : Shape

The computer is saying, “I don’t know how to display data of this type.”
How do we fix the stupid computer?
A More Interesting Type (cont.)

Add a line “deriving Show” at the end of the type declaration:

```haskell
data Shape = Rectangle Float Float
            | Ellipse Float Float
    deriving Show
```

This tells the computer, “just display data of this type naively.”

Now you can enter:

```
Rectangle 1.0 2.0
```

And the computer will display it.
A More Interesting Function

Let us write a function to compute areas of shapes.

\[
\text{area :: Shape} \to \text{Float}
\]

- Area of rectangle is width times height.
  \[
  \text{area (Rectangle w h)} = w \times h
  \]
  The parentheses are needed when there are parameters to the constructor.

- Area of ellipse is \( \pi \) times width times height.
  \[
  \text{area (Ellipse w h)} = \pi \times w \times h
  \]

- Done!
Constructor vs Function

Consider again:

\[ \text{Rectangle} \ 1.0 \ 2.0 :: \ Shape \]

- The constructor is acting like a function:
  \[ \text{Rectangle} :: \ Float \to \ Float \to \ Shape \]
  
  In fact you can use it as such.

- So Red, Green, Blue are like functions requiring no parameters.

- But constructors and functions are different. E.g., cannot use functions
  in pattern matching.
An Introduction to Lists

Some example lists:

[False, True, False] :: [Bool]
[Rectangle 1.0 2.0, Ellipse 2.0 3.0] :: [Shape]
[] -- the empty list, pronounced nil

• We will discuss the type of [] later. For now, it just works.

• Because of strong typing, Haskell lists are \textit{homogeneous}: all elements in a list must be of the same type.

To simulate heterogenous lists, use list of a union type, just like how we mix rectangles and ellipses in the same list.
An Introduction to Lists (cont.)

- The operator : adds an element to the front of a list.
  
  False:[True] gives [False, True]

- In fact, [] and : are constructors of the list types.
  
  [] :: [Bool]
  (:) :: Bool -> [Bool] -> [Bool]
  
  So you can use them in pattern matching.

- [False, True] is really constructed in these stages:
  
  1. start with constructor []
  2. use constructor : to add True. True:[]
  3. use constructor : to add False. False:True:[]
A Function Of List

Write a function that adds up a list of integers.

\[
\text{addList} :: [\text{Integer}] \rightarrow \text{Integer}
\]

- Hey I know how to do it when the list is empty.
  \[
  \text{addList} \; [] = 0
  \]

- If the list is not empty, then it is like \(x:xs\), where \(x\) is the first number and \(xs\) is the rest of the list. I will add \(x\) to the sum of \(xs\).
  The sum of \(xs\) is, of course, \(\text{addList} \; xs\).
  \[
  \text{addList} \; (x:xs) = x + \text{addList} \; xs
  \]

- Done!
A More Interesting Function of List

Write a function that adds up the areas in a list of shapes.

\[
\text{areaList} :: [\text{Shape}] \rightarrow \text{Float}
\]

• Again, I know how to deal with the empty list.

\[
\text{areaList} [] = 0
\]

• If the list is like \(x:xs\), I will compute the area of \(x\), then add it to the sum of the areas in \(xs\).

\[
\text{areaList} (x:xs) = \text{area} \ x + \text{areaList} \ xs
\]

• Done!