Haskell 7

Declaring types and classes

Type Declarations or Type Synonyms

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

String is a synonym for the type [Char].

1

Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

we can define:

```
origin    :: Pos
origin     = (0,0)

left      :: Pos → Pos
left (x,y) = (x-1,y)
```

2

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define:

```
mult      :: Pair Int → Int
mult (m,n) = m*n

copy      :: a → Pair a
copy x     = (x,x)
```

3

Type declarations can be nested:

```
type Pos   = (Int,Int)

type Trans = Pos → Pos
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

4

Data Declarations or Algebraic Data Types

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

5

Note:

- The two values False and True are called the constructors for the type Bool.
- Type and constructor names must begin with an upper-case letter.

6

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers     :: [Answer]
answers      = [Yes,No,Unknown]

flipAns        :: Answer → Answer
flipAns Yes     = No
flipAns No      = Yes
flipAns Unknown = Unknown
```

7

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square          :: Float → Shape
square n         = Rect n n

area            :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

8

Note:

- **Shape** has values of the form **Circle r** where **r** is a float, and **Rect x y** where **x** and **y** are floats.
- **Circle** and **Rect** can be viewed as functions that construct values of type **Shape**:

```
Circle :: Float → Shape

Rect   :: Float → Float → Shape
```

9

```
data Shape = Circle Float
           | Rect Float Float
```

However:

```
Prelude> Circle 3

<interactive>:10:1:
    No instance for (Show Shape) arising
from a use of 'print'
    In a stmt of a interactive GHCi
command: print i
```

print only works for things that derive **Show**

GHCi is trying to call the **print** function

10

```
data Shape = Circle Float
           | Rect Float Float
           deriving (Show)
```

```
Prelude> Circle 3

Circle 3.0
```

11

```
data Shape = Circle Float
           | Rect Float Float
           deriving (Show)
```

```
Prelude> Circle 3 == Circle 4

<interactive>:6:10:
    No instance for (Eq Shape) arising from
a use of '=='
    In the xpression: Circle 3 == Circle 4
    In an ation for 'it': it = Circle 3
== C
```

`==` only works for things that derive **`Eq`**

GHCi is trying to call the `==` function


```
data Answer = Yes | No | Unknown
    deriving (Show)
```

```
flipAns Yes     = No
flipAns No      = Yes
flipAns Unknown = Unknow
```

```
Prelude> flip Yes
No
```

but:

```
flipAns a
  | a == Yes = No
  | a == No  = Yes
  | otherwise = Unknown
```

No instance for (Eq Answer) arising from a use of '==' In the expression: a == Yes


```
data Shape = Circle Float
           | Rect Float Float
           deriving (Show,Eq)
```

```
Prelude> Circle 3 == Circle 4

False
```

What if I want to change the default behavior, saying, for example, that I want **`Rectangle 1 2`** to be equal to **`Rectangle 2 1`**?

Typeclasses

- typeclass **`Eq`** is defined as follows:

```
class Eq a where
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
x == y = not (x /= y)
x /= y = not (x == y)
```

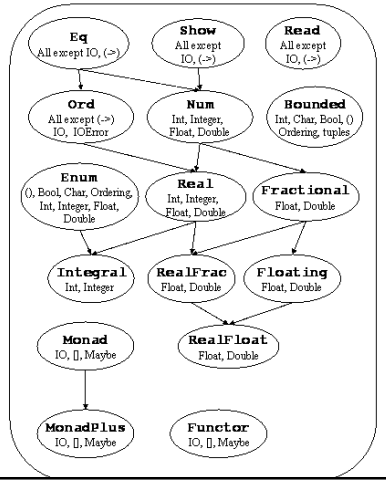
```
data Shape = Circle Float
           | Rect Float Float
           deriving (Show,Eq)

instance Eq Shape where
  Circle x == Circle y = x==y
  Rect x y == Rect a b =
    ((x==a) && (y==b)) || ((x==b) && y==a)
```

```
> Rect 1 2 == Rect 2 1
True
> Rect 1 2 /= Rect 1 2
False
```

16

Standard Haskell Classes



17

Standard Typeclasses

- **Eq** is used for types that support equality testing
  - == and /=
- **Ord** is for types that have an ordering
  - >, <, >= and <=
- Members of **Show** can be presented as strings
  - The most used function that deals with the **Show** typeclass is show
- **Enum** members are sequentially ordered types. The main advantage is that we can use its types in list ranges
- **Bounded** members have an upper and a lower bound.
- **Num** is a numeric typeclass.

18

Not surprisingly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv    :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

19

```
data Point = Point Float Float
```

Data type

Value constructor

- We can also have the same name for the data type and for the value constructor

20

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors
Zero :: Nat and
Succ :: Nat → Nat.

21

Note:

- A value of type **Nat** is either **Zero**, or of the form **Succ n** where **n :: Nat**. That is, **Nat** contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

22

⌘ We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function 1+.

⌘ For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

1 + (1 + (1 + 0)) = 3

23

Using recursion, it is easy to define functions that convert between values of type **`Nat`** and **`Int`**:

```
nat2int          :: Nat → Int
nat2int Zero      = 0
nat2int (Succ n) = 1 + nat2int n

int2nat  :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

24

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add    :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

25

For example:

```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

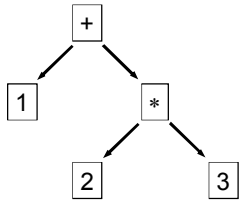
Note:

⌘The recursive definition for add corresponds to the laws 0+n = n and (1+m)+n = 1+(m+n).

26

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



27

Using recursion, a suitable new type to represent such expressions can be declared by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

28

Using recursion, it is now easy to define functions that process expressions. For example:

```
size          :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval          :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

29

Note:

- The three constructors have types:

```
Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr
```

- Many functions on expressions can be defined by replacing the constructors by other functions using a suitable **foldExpr** function. For example:

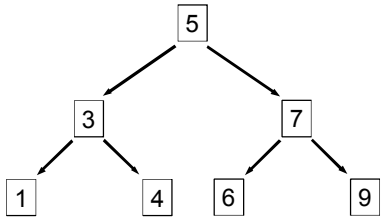
```
eval = foldExpr id (+) (*)
```

Exercise!

30

Binary Trees

In computing, it is often useful to store data in a two-way branching structure or binary tree.



31

Using recursion, a suitable new type to represent such binary trees can be declared by:

```
data Tree a = Leaf a
            | Node (Tree a) a (Tree a)
```

For example, the tree on the previous slide would be represented as follows:

```
t :: Tree Int
t  = Node (Node (Leaf 1) 3 (Leaf 4)) 5
          (Node (Leaf 6) 7 (Leaf 9))
```

32

We can now define a function that decides if a given value occurs in a binary tree:

```
occurs :: Ord a ⇒ a → Tree a → Bool
occurs x (Leaf y)      = x == y
occurs x (Node l y r) = x == y
                        || occurs x l
                        || occurs x r
```

But… in the worst case, when the value does not occur, this function traverses the entire tree.

33

Now consider the function **flatten** that returns the list of all the values contained in a tree:

```
flatten             :: Tree a → [a]
flatten (Leaf x)     = [x]
flatten (Node l x r) = flatten l
                       ++ [x]
                       ++ flatten r
```

A tree is a search tree if it flattens to a list that is ordered. Our example tree is a search tree, as it flattens to the ordered list [1,3,4,5,6,7,9].

34

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
occurs x (Leaf y)                = x == y
occurs x (Node l y r) | x == y = True
                      | x < y  = occurs x l
                      | x > y  = occurs x r
```

This new definition is more efficient, because it only traverses one path down the tree.

35

Getter functions

- How do I get the components?

```
data Point = Point Double Double

xval :: Point -> Double
xval (Point x _) = x

yval :: Point -> Double
yval (Point _ y) = y
```

36

Record Syntax

```
data Point = Point {xval::Double, yval::Double}
```

```
> let p = Point 1 2
> xval p
1.0
```

- Makes code more readable

```
let b = Point {xval = 2, yval = 3}
```

37

Exercises

(1) Using recursion and the function add, define a function that multiplies two natural numbers.

(2) A binary tree is complete if the two sub-trees of every node are of equal size. Define a function that decides if a binary tree is complete.

38

Esercizio

- Definire una funzione di ordine superiore **`foldExpr`** per le espressioni con la quale si possano definire le funzioni **`size`** e **`eval`**
- Grazie alla **`foldExpr`**, invece di definire

```
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

si potrà scrivere:

```
eval expr = foldExpr (+) (*) id
```

- Definire poi la funzione **`size`** tramite la **`foldExpr`**

39

These slides were adapted from the material of the book

Graham Hutton, Programming in Haskell, Cambridge University Press, 2nd edition, 2016

40