L6: Functional and Logic Programming

Overloading and type classes. Algebraic types

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

There are two ways to define a function which works over more than one type:

Polymorphism: A function is polymorphic if it has **a single definition** which works over many types.

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1+ length xs
```

Overloading: A function is overloaded if it has **different definitions** with the same name over a variety of types.

- Addition `(+)` is defined over all numeric types, with different definitions.
- Equality `(==)` is defined over many types, with different definitions.

Why overloading?

```
-- List membership elem without overloading
elemBool :: Bool -> [Bool] -> Bool   -- list of Bool
elemBool _ [] = False
elemBool x (y:ys) = (x ==_Bool y) || elemBool x ys

elemInt :: Int -> [Int] -> Int       -- list of Int
elemInt _ [] = False
elemInt x (y:ys) = (x ==_Int y) || elemInt x ys
```

$==_{\texttt{Bool}}$, $==_{\texttt{Int}}$ are functions with different implementations.

Why overloading?

```
-- List membership  elem  without overloading
elemBool :: Bool -> [Bool] -> Bool   -- list of Bool
elemBool _ [] = False
elemBool x (y:ys) = (x ==Bool y) || elemBool x ys

elemInt :: Int -> [Int] -> Int        -- list of Int
elemInt _ [] = False
elemInt x (y:ys) = (x ==Int y) || elemInt x ys
```

$==_{\texttt{Bool}}$, $==_{\texttt{Int}}$ are functions with different implementations.
With overloading we can define a type class `Eq a`

- all types which are **instances** of `Eq a` have their own implementation of boolean equality `(==)`
- `Bool` and `Int` are instances of type class `Eq a`

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = (x == y) || elem x ys
```

Reuse: The definition of `elem` can be used over all types with equality (that is, types which are instances of type class `Eq a`)

Readability: It is much easier to read $==$ than $==_{\mathtt{Int}}$ and so on.

Reuse: The definition of `elem` can be used over all types with equality (that is, types which are instances of type class `Eq a`)

Readability: It is much easier to read $==$ than $==_{\texttt{Int}}$ and so on.

Haskell allows to define and instantiate type classes.

Reuse: The definition of `elem` can be used over all types with equality (that is, types which are instances of type class `Eq a`)

Readability: It is much easier to read $==$ than $==_{\mathtt{Int}}$ and so on.

Haskell allows to define and instantiate type classes.

1. Defining the equality class:

```
class Eq a where
   (==) :: a -> a -> Bool
```

Reuse: The definition of `elem` can be used over all types with equality (that is, types which are instances of type class `Eq a`)

Readability: It is much easier to read `==` than $==_{\texttt{Int}}$ and so on.

Haskell allows to define and instantiate type classes.

1. Defining the equality class:

```
class Eq a where
  (==) :: a -> a -> Bool
```

2. Defining an instance of the equality class

```
instance Eq Bool where
   True  == True  = True
   False == False = True
   _     == _     = False
```

Defining functions for type classes

Example: Polymorphic functions which use equality

```
-- allEqual lst checks if all elements in lst are equal
allEqual :: Eq a => [a] -> Bool
allEqual []         = True
allEqual [_]        = True
allEqual (x:y:xs) = (x==y) && allEqual (y:xs)

> allEqual [1,1,1]     -- ok
True
> allEqual[(+),(+)]    -- function types are not instances of Eq a
error:
  ...
```

```
-- class definition
class Visible a where
  toString :: a -> String
  size     :: a -> Int
```

`Visible` things can be viewed using the `toString` function. Also, we can get an estimate of their size with the `size` function.

```
-- instantiate Bool to be Visible
instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _         = 1

-- lists of Visible are Visible
instance Visible a => Visible [a] where
  toString = concat . map toString
  size = foldr (+) 1 . map size
```

Built-in type classes

`Eq` and `Ord`

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

Built-in type classes

Eq and Ord

```
class Eq a where
  (==),(/=) :: a -> a -> Bool
  x /= y = not (x==y)     -- default definition
  x == y = not (x/=y)     -- default definition
class Eq a => Ord a where
  (<), (<=), (<=), (>=) :: a -> a -> Bool
  max, min   :: a -> a -> Bool
  compare    :: a -> a -> Ordering
  x <= y = (x < y || x == y)
  x >  y = y < x
  max x y
    | x >= y     = x
    | otherwise = y
  min x y
    | x <= y     = x
    | otherwise = y
```

Built-in type classes

Eq and Ord

```
class Eq a where
  (==),(/=) :: a -> a -> Bool
  x /= y = not (x==y)     -- default definition
  x == y = not (x/=y)     -- default definition
class Eq a => Ord a where
  (<), (<=), (<=), (>=) :: a -> a -> Bool
  max, min   :: a -> a -> Bool
  compare    :: a -> a -> Ordering
  x <= y = (x < y || x == y)
  x >  y = y < x
  max x y
    | x >= y    = x
    | otherwise = y
  min x y
    | x <= y    = x
    | otherwise = y
```

REMARK: Ord is **inheriting** the operations of Eq.

```
ins x []     = [x]
ins x (y:ys)
  | x<= y       = x:(y:ys)
  | otherwise =y:ins x ys
iSort []   = []
iSort (x:xs) = ins x (iSort xs)

> :type ins
ins :: Ord t => t -> [t] -> [t]
> :type iSort
iSort :: Ord a => [a] -> [a]
> iSort [7,1,3,2,9,8,10]
[1,2,3,7,8,9,10]
```

```
ins x []     = [x]
ins x (y:ys)
  | x<= y       = x:(y:ys)
  | otherwise =y:ins x ys
iSort []   = []
iSort (x:xs) = ins x (iSort xs)

> :type ins
ins :: Ord t => t -> [t] -> [t]
> :type iSort
iSort :: Ord a => [a] -> [a]
> iSort [7,1,3,2,9,8,10]
[1,2,3,7,8,9,10]
```

REMARK: Haskell can compute the most general type of `ins` and `iSort`.

```
-- multiple inheritance
class (Ord a, Visible a) => OrdVis a

-- multiple constraints in instance declaration
instance (Eq a,Eq b) => Eq (a,b) where
    (x,y) == (z,w) = x == z && y == w
```

`Enum` types can be used to generate lists like `[2,4,6,8]` using enumeration expressions like `[2,4..8]`. The class definition is

```
class Ord a => Enum a where
  toEnum         :: Int a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]                -- [n .. ]
  enumFromThen   :: a -> a -> [a]           -- [n,m ..]
  enumFromTo     :: a -> a -> [a]           -- [n .. m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n' .. m]
```

More built-in type classes

Enum

`Enum` types can be used to generate lists like `[2,4,6,8]` using enumeration expressions like `[2,4..8]`. The class definition is

```
class Ord a => Enum a where
  toEnum         :: Int a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]              -- [n .. ]
  enumFromThen   :: a -> a -> [a]         -- [n,m ..]
  enumFromTo     :: a -> a -> [a]         -- [n .. m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n' .. m]
```

- `Char` and `Int` are instances of `Enum`

More built-in type classes

Enum

`Enum` types can be used to generate lists like `[2,4,6,8]` using enumeration expressions like `[2,4..8]`. The class definition is

```
class Ord a => Enum a where
  toEnum         :: Int a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]             -- [n .. ]
  enumFromThen   :: a -> a -> [a]        -- [n,m ..]
  enumFromTo     :: a -> a -> [a]        -- [n .. m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n' .. m]
```

- `Char` and `Int` are instances of `Enum`
- Examples of built-in functions defined on `Enum` types:

```
succ,pred :: Enum a => a -> a -- successor and predecessor
succ = toEnum . (+1) . fromEnum
pred = toEnum . (subtract 1) . fromEnum
```

More built-in type classes

Show

`Show` is a type class for types whose values can be written as strings.

```
type ShowS = String -> String
showsPrec :: Int -> a -> ShowS
show      :: a -> String
showList  :: [a] -> ShowS
```

`Show` is a type class for types whose values can be written as strings.

```
type ShowS = String -> String
showsPrec :: Int -> a -> ShowS
show      :: a -> String
showList  :: [a] -> ShowS
```

Possible instance declarations might be

```
instance (Show a,Show b) => Show (a,b) where
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

- Basic types: `Int`, `Integer`, `Float`, `Doyble`, `Bool`, `Char`
- Composite types:
  - tuple types $(\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n)$
  - list types $[\mathtt{T}_1]$
  - function types $(\mathtt{T}_1 \mathtt{->} \mathtt{T}_2)$ where $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$ are themselves types.

- Basic types: `Int`, `Integer`, `Float`, `Doyble`, `Bool`, `Char`
- Composite types:
  - tuple types $(T_1, T_2, \ldots, T_n)$
  - list types $[T_1]$
  - function types $(T_1 \text{->} T_2)$ where $T_1, T_2, \ldots, T_n$ are themselves types.

In Haskell, programmers can define their own data types, with the `data` construct (see next slide).

1. A data type for numeric trees:

   ```
   data NTree = NilT
              | Node Integer NTree NTree
   ```

   This `data` declaration defines two things:

   1. A type constructor: `NTree`
   2. Two data constructors: `NilT` (for the empty tree) and `Node` for a tree with two subtrees.

   Note: `NTree` is a recursive type.

1. A data type for numeric trees:

```
data NTree = NilT
           | Node Integer NTree NTree
```

   This `data` declaration defines two things:

   1. A type constructor: `NTree`
   2. Two data constructors: `NilT` (for the empty tree) and `Node` for a tree with two subtrees.

   Note: `NTree` is a recursive type.

2. The predefined `Maybe` type – used in modeling program errors:

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

   Note: `NTree` is a polymorphic type.

Algebraic type definitions

$$
\begin{array}{l}
\texttt{data Typename} \\
\quad = \texttt{Con}_1\ \texttt{T}_{1,1}\ \ldots\ \texttt{T}_{1,k_1} \\
\quad \mid\ \texttt{Con}_2\ \texttt{T}_{2,1}\ \ldots\ \texttt{T}_{2,k_2} \\
\quad\quad \ldots \\
\quad \mid\ \texttt{Con}_n\ \texttt{T}_{n,1}\ \ldots\ \texttt{T}_{n,k_n}
\end{array}
$$

- Each data constructor $\texttt{Con}_i$ is followed by $\texttt{k}_i$ types. We build elements of type `Typename` by applying these data constructors to arguments of the types given in the definition , so that

  $$\texttt{Con}_i\ \texttt{v}_{i,1}\ \ldots\ \texttt{v}_{i,k_i}$$

  will be a member of the type `Typename` if $\texttt{v}_{i,j}$ is of type $\texttt{T}_{i,j}$ for $1 \leq j \leq k_i$.

Note: The `data` declaration defines every $\texttt{Con}_i$ as a function with the type

$$\texttt{Con}_i\ ::\ \texttt{T}_{i,1} -> \ \ldots\ \texttt{T}_{i,k_i}\ \texttt{->}\ \texttt{Typename}$$

A algebraic type for geometric shapes

```
data Shape = Circle Float | Rectangle Float Float
```

- Definitions over algebraic types use pattern matching both to distinguish between different alternatives and to extract components from particular elements:

```
area :: Shape -> Float
area (Circle r)        = pi*r*r
area (Rectangle a b) = a*b
```

A algebraic type for geometric shapes

```
data Shape = Circle Float | Rectangle Float Float
```

- Definitions over algebraic types use pattern matching both to distinguish between different alternatives and to extract components from particular elements:

```
area :: Shape -> Float
area (Circle r)        = pi*r*r
area (Rectangle a b) = a*b
```

- When we introduce a ned algebraic type, we can `derive` instances of built-in type classes, including `Eq`, `Ord`, `Enum`, `Show`, and `Read`.

  Example:

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq,Ord,Enum,Show,Read)
```

  E.g., we can write `[Spring..Autumn]` instead of `[Spring,Summer,Autumn]`.

```
data Expr = Lit Integer |
            Add Expr Expr |
            Sub Expr Expr
```

Given an expression, we might want to

1. evaluate it (`eval`)
2. turn it into a string, which is then printed
3. estimate its size: how many operators does it have?

```
eval :: Expr -> Integer
show :: Expr -> String
size :: Expr -> Integer
...
```

```
data NTree = NilT | Node Integer NTree NTree
```

Define the functions

```
sumtree, depth :: NTree -> Integer
occurs :: NTree -> Integer -> Integer
```

such that

- `sumtree nt` returns the sum of numbers in `nt`
- `depth nt` returns the depth of]tt nt. For example, `depth NilT` must be 0.
- `occurs nt p` returns how many times number `p` occurs in `nt`.

Addition of integers is associative $\Rightarrow$ we may want to write a program that turns expressions into right bracketed form, as shown in the following table:

Initial expression	Right bracketed result
$(2+3)+4$	$2+(3+4)$
$((2+3)+4)+5$	$2+(3+(4+5))$
$((2-((6+7)+8))+4)+5$	$(2-(6+(7+8)))+(4+5)$

Define a recursive function `rassoc::Expr->Expr` that does this transformation.

First attempt

```
rassoc :: Expr->Expr
rassoc (Lit n) = Lit n
rassoc (Add (Add e1 e2) e3) = Add e1 (Add e2 e3)
rassoc (Add e1 e2) = Add (rassoc e1) (rassoc e2)
rassoc (Sub e1 e2) = Sub (rassoc e1) (rassoc e2)
```

Recursive types

Quiz: rearranging expressions (continued)

First attempt

```
rassoc :: Expr->Expr
rassoc (Lit n) = Lit n
rassoc (Add (Add e1 e2) e3) = Add e1 (Add e2 e3)
rassoc (Add e1 e2) = Add (rassoc e1) (rassoc e2)
rassoc (Sub e1 e2) = Sub (rassoc e1) (rassoc e2)
```

1. Is this definition doing the desired transformation? Why/why not?

First attempt

```
rassoc :: Expr->Expr
rassoc (Lit n) = Lit n
rassoc (Add (Add e1 e2) e3) = Add e1 (Add e2 e3)
rassoc (Add e1 e2) = Add (rassoc e1) (rassoc e2)
rassoc (Sub e1 e2) = Sub (rassoc e1) (rassoc e2)
```

1. Is this definition doing the desired transformation? Why/why not?
2. Find a better implementation (Second attempt).

Polymorphic algebraic types

Algebraic type definitions can contain type variables a, b and so on, defining polymorphic types. The definitions are as before, with the type variables used in the definition appearing after the type name on the left side of the definition.

Example (Polymorphic pairs)

```
data Pairs a = Pr a a
```

Then

```
Pr True False :: Pairs Bool
Pr [] [3] :: Pairs [Int]
Pr [] [] :: Pairs [a]
```

We can define

```
equalPair :: Eq a => Pairs a -> Bool
aqualPair (Pr x y) = (x==y)
```

```
data Tree a = Nil | Node a (Tree a) (Tree a)
              deriving (Eq,Ord,Show,Read)
```

- Elements have arbitrary type `a`.
- The definitions of `depth` and `occurs` from `NTree` remain unchanged for `(Tree a)`.

References

Chapter 13: *Overloading, type classes and type checking* and Chapter 14: *Algebraic types* from

Simon Thompson: *Haskell: The Craft of Functional Programming.* Second edition. Pearson Addison Wesley. 1999.