CSCE 314
Programming Languages
Haskell: Declaring Types and Classes
Dr. Hyunyoung Lee
Outline

- Declaring Data Types
- Class and Instance Declarations
Defining New Types

Three constructs for defining types:

1. data – Define a new algebraic data type from scratch, describing its constructors

2. type – Define a synonym for an existing type (like typedef in C)

3. newtype – A restricted form of data that is more efficient when it fits (if the type has exactly one constructor with exactly one field inside it). Used for defining “wrapper” types
Data Declarations

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.

- The two values False and True are called the constructors for the data type Bool.
- Type and constructor names must begin with an upper-case letter.
- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

More examples from standard Prelude:

```
data () = () -- unit datatype
data Char = ... | 'a' | 'b' | ...
```
Values of new types can be used in the same ways as those of built in types. For example, given

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

we can define:

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

```haskell
flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```

Constructors construct values, or serve as patterns.
Another example:

```haskell
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Constructors construct values, or serve as **patterns**

```haskell
next :: Weekday -> Weekday
next Mon = Tue
next Tue = Wed
next Wed = Thu
next Thu = Fri
next Fri = Sat
next Sat = Sun
next Sun = Mon

workDay :: Weekday -> Bool
workDay Sat = False
workDay Sun = False
workDay _   = True
```
Constructors with Arguments

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

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

we can define:

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

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

- 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:

```haskell
Circle :: Float → Shape
Rect  :: Float → Float → Shape
```
Another example:

```haskell
data Person = Person Name Gender Age

data Gender = Male | Female

type Name = String

type Age = Int
```

With just one constructor in a data type, often the constructor is named the same as the type (cf. Person). Now we can do:

```haskell
let x = Person "Jerry" Female 12
    y = Person "Tom" Male 12
in ...
```

Quiz: What are the types of the constructors Male and Person?

- `Male :: Gender`
- `Person :: Name -> Gender -> Age -> Person`
Pattern Matching

name (Person n _ _) = n

oldMan (Person _ Male a) | a > 100 = True
oldMan (Person _ _ _) = False

> let yoda = Person “Yoda” Male 999
  in oldMan yoda
True

findPrsn n (p@(Person m _ _):ps)
  | n == m = p
  | otherwise = findPrsn n ps

> findPrsn “Tom”
  [Person “Yoda” Male 999, Person “Tom” Male 7]
Person “Tom” Male 7
Parameterized Data Declarations

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

```
data Pair a b = Pair a b
```

we can define:

```hs
x = Pair 1 2
y = Pair "Howdy" 42
```

```hs
 first :: Pair a b -> a
 first (Pair x _) = x
```

```hs
 apply :: (a->a') -> (b->b') -> Pair a b -> Pair a' b'
 apply f g (Pair x y) = Pair (f x) (g y)
```
Another example:
Maybe type holds a value (of any type) or holds nothing

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

a is a type parameter, can be bound to any type

```
Just True :: Maybe Bool
Just "x" :: Maybe [Char]
Nothing :: Maybe 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)
```
Type Declarations

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]`.

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)
```
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)
```
Type declarations can be nested:

```plaintext
type Pos = (Int, Int)
type Trans = Pos -> Pos
```

However, they cannot be recursive:

```plaintext
type Tree = (Int, [Tree])
```
Recursive Data Types

New types can be declared in terms of themselves. That is, data types can be recursive.

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

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)
- ...

Example function:

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
```
Parameterized Recursive Data Types - Lists

data List a = Nil | Cons a (List a)

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

> sum Nil
0
> sum (Cons 1 (Cons 2 (Cons 2 Nil)))
5
Trees

A binary Tree is either Tnil, or a Node with a value of type a and two subtrees (of type Tree a)

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

Find an element:

```
treeElem :: (a -> Bool) -> Tree a -> Maybe a

treeElem p Tnil = Nothing

treeElem p t@(Node v left right)
  | p v = Just v
  | otherwise = treeElem p left `combine` treeElem p right

where combine (Just v) r = Just v
      combine Nothing r  = r
```

Compute the depth:

```
depth Tnil = 0

depth (Node _ left right) = 1 +
  (max (depth left) (depth right))
```
Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.
Using recursion, a suitable new type to represent such expressions can be declared by:

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

\[
\begin{align*}
\text{size} & : \text{Expr} \rightarrow \text{Int} \\
\text{size} (\text{Val} \ n) & = 1 \\
\text{size} (\text{Add} \ x \ y) & = \text{size} \ x + \text{size} \ y \\
\text{size} (\text{Mul} \ x \ y) & = \text{size} \ x + \text{size} \ y \\
\text{eval} & : \text{Expr} \rightarrow \text{Int} \\
\text{eval} (\text{Val} \ n) & = n \\
\text{eval} (\text{Add} \ x \ y) & = \text{eval} \ x + \text{eval} \ y \\
\text{eval} (\text{Mul} \ x \ y) & = \text{eval} \ x \ast \text{eval} \ y
\end{align*}
\]
Note:

- The three constructors have types:
  
  \[
  \text{Val} :: \text{Int} \to \text{Expr} \\
  \text{Add} :: \text{Expr} \to \text{Expr} \to \text{Expr} \\
  \text{Mul} :: \text{Expr} \to \text{Expr} \to \text{Expr}
  \]

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

\[
\text{fold} :: (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int} \to \text{Int}) \to \\
(\text{Int} \to \text{Int} \to \text{Int}) \to \text{Expr} \to \text{Int} \\
\text{fold } f \ g \ h \ (\text{Val } n) = f \ n \\
\text{fold } f \ g \ h \ (\text{Add } a \ b) = g \ (\text{fold } f \ g \ h \ a) \ (\text{fold } f \ g \ h \ b) \\
\text{fold } f \ g \ h \ (\text{Mul } a \ b) = h \ (\text{fold } f \ g \ h \ a) \ (\text{fold } f \ g \ h \ b)
\]

\[
\text{eval} = \text{fold } \text{id} \ (+) \ (*)
\]
About Folds

A fold operation for Trees:

treeFold :: t -> (a -> t -> t -> t) -> Tree a -> t

\[
treeFold \ f \ g \ Tnil = f \\
treeFold \ f \ g \ (Node \ x \ l \ r) = g \ x \ (treeFold \ f \ g \ l) \ (treeFold \ f \ g \ r)
\]

How? Replace all \text{\texttt{Tnil}} constructors with \(f\), all \text{\texttt{Node}} constructors with \(g\). Examples:

\[
> \text{let } tt = \text{Node } 1 \ (\text{Node } 2 \ Tnil \ Tnil) \\
\hspace{1cm} (\text{Node } 3 \ Tnil \ (\text{Node } 4 \ Tnil \ Tnil)) \\
> \text{treeFold } 1 \ (\lambda x \ y \ z \rightarrow 1 + \max y z) \ tt \\
4 \\
> \text{treeFold } 1 \ (\lambda x \ y \ z \rightarrow x \ast y \ast z) \ tt \\
24 \\
> \text{treeFold } 0 \ (\lambda x \ y \ z \rightarrow x + y + z) \ tt \\
10
\]
Deriving

• Experimenting with the above definitions will give you many errors
• Data types come with no functionality by default, you cannot, e.g., compare for equality, print (show) values etc.
• Real definition of Bool

\[
data \text{Bool} = \text{False} \mid \text{True}
\]
\[
\text{deriving (Eq, Ord, Enum, Read, Show, Bounded)}
\]
• A few standard type classes can be listed in a `deriving` clause
• Implementations for the necessary functions to make a data type an instance of those classes are generated by the compiler
• `deriving` can be considered a shortcut, we will discuss the general mechanism later
Exercises

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

(2) Define a suitable function fold for expressions, and give a few examples of its use.

(3) 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.
Outline

- Declaring Data Types
- Class and Instance Declarations
Type Classes

- A new class can be declared using the `class` construct.
- Type classes are classes of types, thus not types themselves.
- Example:

  ```haskell
  class Eq a where
  (==), (=/=) :: a -> a -> Bool
  -- Minimal complete definition: (==) and (=/=)
  x /= y   = not (x == y)
  x == y   = not (x /= y)
  ```

- For a type `a` to be an instance of the class `Eq`, it must support equality and inequality operators of the specified types.
- Definitions are given in an instance declaration.
- A class can specify default definitions.
Instance Declarations

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

Let us make Bool be a member of Eq

instance Eq Bool where
   (==) False False  = True
   (==) True True    = True
   (==) _ _          = False

- Due to the default definition, (=/=) need not be defined
- deriving Eq would generate an equivalent definition
Showable Weekdays

class Show a where
    showsPrec :: Int -> a -> ShowS -- to control parenthesizing
    show :: a -> String

    showsPrec _ x s = show x ++ s
    show x            = showsPrec 0 x ""

showsPrec can improve efficiency: (((as ++ bs) ++ cs) ++ ds)
vs. (as ++) . (bs ++) . (cs ++) . (ds ++)

Option 1:

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving Show

> map show [Mon, Tue, Wed]
[“Mon”, “Tue”, “Wed”]
Showable Weekdays

Option 2:

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

instance Show Weekday where
  show Mon = “Monday”
  show Tue = “Tuesday”
  ...

> map show [Mon, Tue, Wed]
[“Monday”, “Tuesday”, “Wednesday”]
Parameterized Instance Declarations

Every list is showable if its elements are

```haskell
instance Show a => Show [a] where
    show []     = "[]"
    show (x:xs) = "[" ++ show x ++ showRest xs
                   where showRest []     = ""]"
                   showRest (x:xs) = "," ++ show x ++ showRest xs
```

Now this works:

```haskell
> show [Mon, Tue, Wed]
"[Monday,Tuesday,Wednesday]"
```
Showable, Readable, and Comparable Weekdays

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

    deriving (Show, Read, Eq, Ord, Bounded, Enum)

*Main> show Wed
"Wed"
*Main> read "Fri" :: Weekday
Fri
*Main> Sat == Sun
False
*Main> Sat == Sat
True
*Main> Mon < Tue
True
*Main> Tue < Tue
False
*Main> Wed `compare` Thu
LT
Bounded and Enumerable Weekdays

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Show, Read, Eq, Ord, Bounded, Enum)

*Main> minBound :: Weekday
  Mon
*Main> maxBound :: Weekday
  Sun
*Main> succ Mon
  Tue
*Main> pred Fri
  Thu
*Main> [Fri .. Sun]
  [Fri,Sat,Sun]
*Main> [minBound .. maxBound] :: [Weekday]
  [Mon,Tue,Wed,Thu,Fri,Sat,Sun]