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
```

Boo 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

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

we can define:

```
answers :: [Answer]
answers = [Yes, No, Unknown]
flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```
Another example:

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

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

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
```
Constructors with Arguments
The constructors in a data declaration can also have parameters. For example, given

\[
data\ Shape = \text{Circle} \ Float \mid \text{Rect} \ Float \ Float \ Float
\]

we can define:

\[
\begin{align*}
\text{square} & : \ Float \to \text{Shape} \\
\text{square} \ n & = \text{Rect} \ n \ n \\
\text{area} & : \text{Shape} \to \Float \\
\text{area} \ (\text{Circle} \ r) & = \pi * r^2 \\
\text{area} \ (\text{Rect} \ x \ y) & = x * y
\end{align*}
\]

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

\[
\begin{align*}
\text{Circle} & : \Float \to \text{Shape} \\
\text{Rect} & : \Float \to \Float \to \text{Shape}
\end{align*}
\]
Another example:

```haskell
data Person = Person Name Gender Age

type Name = String

data Gender = Male | Female

type Age = Int
```

With just one constructor in a data type, often 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:

\[
x = \text{Pair} \ 1 \ 2
\]
\[
y = \text{Pair} \ "\text{Howdy}" \ 42
\]

\[
\text{first} :: \text{Pair} \ a \ b \rightarrow a
\]
\[
\text{first} \ (\text{Pair} \ x \ _) = x
\]

\[
\text{apply} :: (a \rightarrow a') \rightarrow (b \rightarrow b') \rightarrow \text{Pair} \ a \ b \rightarrow \text{Pair} \ a' \ b'
\]
\[
\text{apply} \ f \ g \ (\text{Pair} \ x \ y) = \text{Pair} \ (f \ x) \ (g \ y)
\]
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
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat -> 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:

\[
\text{treeElem} :: (a \rightarrow \text{Bool}) \rightarrow \text{Tree a} \rightarrow \text{Maybe a} \\
\text{treeElem} p \ Tnil = \text{Nothing} \\
\text{treeElem} p \ (\text{Node} \ v \ \text{left} \ \text{right}) \\
\quad | \ p \ v = \text{Just} \ v \\
\quad | \ \text{otherwise} = \text{treeElem} p \ \text{left} \ `\text{combine}` \ \text{treeElem} p \ \text{right} \\
\quad \text{where} \ \text{combine} (\text{Just} \ v) \ r = \text{Just} \ v \\
\quad \text{combine} \ \text{Nothing} \ r = r
\]

Compute the depth:

\[
\text{depth} \ \text{Tnil} = 0 \\
\text{depth} (\text{Node} \ _ \ \text{left} \ \text{right}) = 1 + \\
(\text{max} (\text{depth} \ \text{left}) (\text{depth} \ \text{right}))
\]
Arithmetic Expressions

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

\[ 1 + 2 \times 3 \]
Using recursion, a suitable new type to represent such expressions can be declared by:

\[
data \text{ Expr } = \text{ Val } \text{ Int} \\
| \text{ Add } \text{ Expr } \text{ Expr} \\
| \text{ Mul } \text{ Expr } \text{ Expr}
\]

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

\[
\text{Add } (\text{Val } 1) (\text{Mul } (\text{Val } 2) (\text{Val } 3))
\]
Using recursion, it is now easy to define functions that process expressions. For example:

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

\[
\begin{align*}
\text{eval} & : \text{Expr} \rightarrow \text{Int} \\
\text{eval} \ (\text{Val} \ n) & = n \\
\text{eval} \ (\text{Add} \ x \ y) & = \text{eval} \ x + \text{eval} \ y \\
\text{eval} \ (\text{Mul} \ x \ y) & = \text{eval} \ x \times \text{eval} \ y
\end{align*}
\]
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 `fold` function. For example:

  ```haskell
  fold :: (Int->Int)->(Int->Int->Int)->
       (Int->Int->Int)->Expr->Int
  fold f g h (Val n) = f n
  fold f g h (Add a b) = g (fold f g h a) (fold f g h b)
  fold f g h (Mul a b) = h (fold f g h a) (fold f g h b)
  eval = fold id (+) (*)
  ```
About Folds

A fold operation for Trees:

\[
\text{treeFold} :: t -> (a -> t -> t -> t) -> \text{Tree} a -> t
\]
\[
\text{treeFold} f g \text{Tnil} = f
\]
\[
\text{treeFold} f g \text{(Node} x \text{l r)}
\]
\[
= g x (\text{treeFold} f g \text{l}) (\text{treeFold} f g \text{r})
\]

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

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

\[
data \text{Bool} = \text{False} | \text{True}
\]

\[
\text{deriving (Eq, Ord, Enum, Read, Show, Bounded)}
\]

• A few standard type classes can be listed in a \underline{deriving} clause
• Implementations for the necessary functions to make a data type an instance of those classes are generated by the compiler
• \underline{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 dering 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”]
Every list is showable if its elements are

```
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:

```
> 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 Prelude.== Sun
False

*Main> Sat Prelude.== 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]