

CSc 372

Comparative Programming Languages

14 : Haskell — Data Types

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

User-defined Datatypes

- Haskell lets us create new datatypes:

$$\text{data } \mathit{Datatype} \ a_1 \ \dots \ a_n = \mathit{constr}_1 \ | \ \dots \ | \ \mathit{constr}_m$$

where

- 1 $\mathit{Datatype}$ is the name of a new type constructor
 - 2 a_1, \dots, a_n are type variables representing the arguments of $\mathit{Datatype}$
 - 3 $\mathit{constr}_1, \dots, \mathit{constr}_m$ are the different ways in which we can create new elements of the new datatype.
- Each constr is of the form

$$\mathit{Name} \ \mathit{type}_1 \ \dots \ \mathit{type}_r$$

where Name is a new name beginning with a capital letter.

Like Enumerations!

- The following definition introduces a new type `Day` with elements `Sun`, `Mon`, `Tue`,...:

```
data Day = Sun|Mon|Tue|Wed|Thu|Fri|Sat
```

- Simple functions manipulating elements of type `Day` can be defined using pattern matching:

```
what_shall_I_do Sun = "relax"  
what_shall_I_do Sat = "go shopping"  
what_shall_I_do _   = "go to work"
```

Like Enumerations — with arguments!

- We can represent temperatures either using centigrade or fahrenheit:

```
data Temp = Centigrade Float |
           Fahrenheit Float
           deriving Show
```

```
freezing      :: Temp -> Bool
freezing (Centigrade temp) = temp <= 0.0
freezing (Fahrenheit temp) = temp <= 32.0
```

- We add the syntax deriving Show so that we can print out elements of the datatype:

```
> Centigrade 66
Centigrade 66.0
```

Recursive Datatypes

- We can define recursive datatypes.
- In fact, we can use datatypes to define our own kind of lists!
- Here's a list of integers:

```
data IntList =  
    IntCons Int IntList |  
    IntNil  
    deriving Show
```

- As usual, a list is either Nil or a Cons cell consisting of an integer and the rest of the list.
- Here's the list [5,6] in our new representation:

```
IntCons 5 (IntCons 6 IntNil)
```

Polymorphic Recursive Datatypes

- Here's a recursive definition of a polymorphic list:

```
data List a =  
  Cons a (List a) |  
  Nil deriving Show
```

- We can define our own versions of head and tail:

```
hd Nil = error "Head of Nil"  
hd (Cons a _) = a
```

```
tl Nil = error "Tail of Nil"  
tl (Cons _ b) = b
```

- And we can construct lists of arbitrary types and take them apart:

```
> hd (tl (Cons 1 (Cons 2 Nil)))  
2
```

```
> hd (tl (Cons "hello" (Cons "bye" Nil)))  
"bye"
```

Polymorphic Binary Tree

- Here's the definition of a binary tree with data in each leaf and internal node:

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

- For example, here's a binary search tree with the elements f, 10, 12, 15, 16:

```
Node  
  (Leaf 5)  
  10  
  (Node  
    (Leaf 12)  
    15  
    (Leaf 16)  
  )
```

Polymorphic Binary Search Tree

- Here's a function that looks up a value in a tree:

```
treemem :: Ord a => Tree a -> a -> Bool
treemem (Leaf v) x = x == v
treemem (Node l v r) x
    | x == v = True
    | x < v = treemem l x
    | x > v = treemem r x
```

- Examples:

```
> let t = Node (Leaf 5) 10 (Node (Leaf 12) 15 (Leaf 16))
> treemem t 16
True
> treemem t 5
True
> treemem t 1
False
```


Homework 1

- Write the function `depth` which calculates the depth of a tree, `leaves` which returns the leaves of a tree, and `inorder` which returns a list of the nodes of the tree in inorder:

```
depth :: Tree a -> Int
```

```
leaves :: Tree a -> [a]
```

```
inorder :: Tree a -> [a]
```

Homework 1...

- Examples:

```
> let t1 = Node (Leaf 5) 10 (Leaf 15)
```

```
> let t2 = Node (Leaf 5) 10 (Node (Leaf 12) 15 (Leaf 16))
```

```
> depth t1
```

```
2
```

```
> depth t2
```

```
3
```

```
> leaves t1
```

```
[5,15]
```

```
> leaves t2
```

```
[5,12,16]
```

```
> inorder t1
```

```
[5,10,15]
```

```
> inorder t2
```

```
[5,10,12,15,16]
```

Homework 2

- Here's a datatype for arithmetic expressions:

```
data Expr = Val Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Neg Expr
          deriving Show
```

- Write a function `eval e` which evaluates an arithmetic expression `e`:

```
eval :: Expr -> Int
```

Homework 2...

- Examples:

```
> eval (Val 5)
```

```
5
```

```
> eval (Add (Val 6) (Val 5))
```

```
11
```

```
> eval (Add (Mul (Val 7) (Val 5)) (Val 7))
```

```
42
```