Sum Types

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Describe the syntax for declaring disjoint data types in Haskell.
- Show how to use disjoint types to represent lists, expressions, and exceptions.
- Explain the operation and implementation of the list, `Maybe` and `Either` data types.
- Use a disjoint datatype to represent an arithmetic calculation.

Simple Type Definitions

Disjoint Type Syntax

`data` *TName* = *CName* [*type* ···] [| *CName* [*type* ···] ···]

A *sum type* has three components: a *name*, a set of `constructors`, and possible *arguments*.

```
1 data Contest = Rock | Scissors | Paper
2 data Velocity = MetersPerSecond Float
3               | FeetPerSecond Float
4 data List a = Cons a (List a)
5             | Nil
6 data Tree a = Node a (Tree a) (Tree a)
7             | Empty
```


Example of `Contest` and `Velocity`

```
1 winner Rock Scissors = "Player 1"
2 winner Scissors Paper = "Player 1"
3 winner Paper Rock = "Player 1"
4 winner Scissors Rock = "Player 2"
5 winner Paper Scissors = "Player 2"
6 winner Rock Paper = "Player 2"
7 winner _ _ = "Tie"
8
9 thrust (FeetPerSecond x) = x / 3.28
10 thrust (MetersPerSecond x)  = x
```

The Most Fun Datatypes Are Recursive

Our Own List Construct

```
data List = Cons Int List
          | Nil
  deriving Show
insertSorted a Nil = Cons a Nil
insertSorted a (Cons b bs)
     | a < b     = Cons a (Cons b bs)
     | otherwise = Cons b (insertSorted a bs)
```

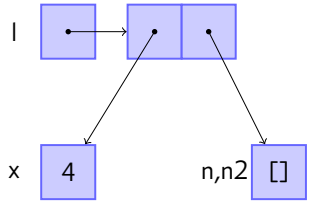
We can run it like this:

```
*Main> let l1 = insertSorted 3 (Cons 2 (Cons 4 Nil))
*Main> l1
Cons 2 (Cons 3 (Cons 4 Nil))
```


Type Constructors and Memory

- When a type constructor is invoked, it causes memory to be allocated.
  - Writing an integer
  - Writing [] or Nil
  - Using : or Cons
- Writing down a variable does not cause memory to be allocated.

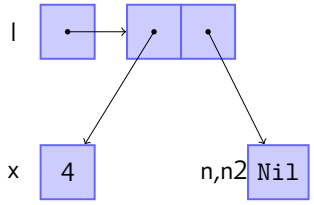
```
x = 4   -- allocates 4
n = []  -- allocates empty list
n2 = n  -- does NOT allocate memory
l = x:n -- A cons cell is allocated, but not the 4 or the empty list
```



Similarly ...

```
x = 4
n = Nil
n2 = n
l = Cons x n
```

- Our own types do the same thing.



Parameters

HASKELL supports *parametric polymorphism*, like templates in C++ or generics in JAVA.

Parametric Polymorphism

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

```
x1 = Cons 1 (Cons 2 (Cons 4 Nil)) -- List Int
x2 = Cons "hi" (Cons "there" Nil) -- List String
x3 = Cons Nil (Cons (Cons 5 Nil) Nil) -- List (List Int)
```

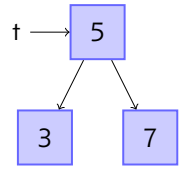
BST Add

- Here is some code for BST Add!
- Note the dual use of a constructor: both for building and for pattern matching.

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
add_bst :: Integer -> Tree Integer -> Tree Integer
add_bst i Empty = Node i Empty Empty
add_bst i (Node x left right)
   | i <= x    = Node x (add_bst i left) right
   | otherwise = Node x left (add_bst i right)
```

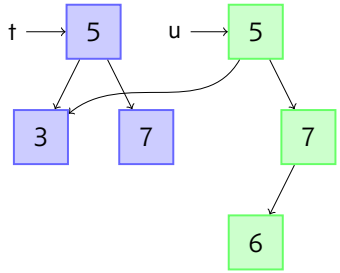

Functional Updating

- It is important to understand functional updating.
- We don't update in place. We make copies, and share whatever we can.
  - Example: add 5,3,7 to a tree `t`
  - `let u = add t 6`
  - `let v = add u 1`



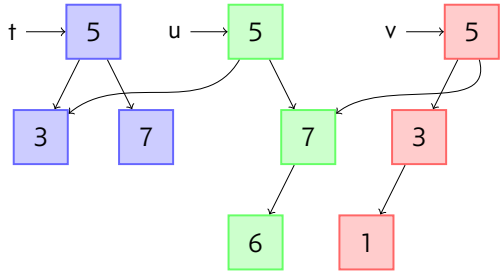
Functional Updating

- It is important to understand functional updating.
- We don't update in place. We make copies, and share whatever we can.
  - Example: add 5,3,7 to a tree `t`
  - `let u = add t 6`
  - `let v = add u 1`



Functional Updating

- It is important to understand functional updating.
- We don't update in place. We make copies, and share whatever we can.
  - Example: add 5,3,7 to a tree `t`
  - `let u = add t 6`
  - `let v = add u 1`



The Maybe Type

The Maybe Type

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

Remember the lookup function that didn't know what to do if the item wasn't in the list?

```
getItem key [] = Nothing
getItem key ((k,v):xs) =
    if key == k then Just v
               else getItem key xs
```

Example:

```
*Main> getItem 3 [(2,"french hens"), (3,"turtle doves")]
Just "turtle doves"
*Main> getItem 5 [(2,"french hens"), (3,"turtle doves")]
Nothing
```


You try!

```
data Tree a = Branch a (Tree a) (Tree a)
            | Empty
  deriving Show
```

1. Write `add :: Tree a -> a -> Tree a`
2. Write `find :: Tree a -> a -> Bool`
3. Write `lookup :: Tree (k,v) -> k -> Maybe v`
4. Write `delete :: Tree a -> a -> Tree a`

The Either Type

The Either Type

```
data Either a b = Left a | Right b
```

We can use it in places where we want to return something, or else an error message.

```
getItem key [] = Left "Key not found"
getItem key ((k,v):xs) =
    if key == k then Right v
               else getItem key xs
```

Example:

```
*Main> getItem 3 [(2,"french hens"), (3,"turtle doves")]
Right "turtle doves"
*Main> getItem 5 [(2,"french hens"), (3,"turtle doves")]
Left "Key not found"
```