Disjoint Types

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

- Describe the syntax for disjoint data types in Haskell.
- Describe a few use-cases for them.

Simple Type Definitions

Disjoint Type Syntax

$$\texttt{data}\ \mathit{Name} = \mathit{Name}\ [\mathit{type} \cdots]\ [\mid \mathit{Name}\ [\mathit{type} \cdots] \cdots]$$

- Note: Constructor names must be capitalized.
- Constructor names also must be unique.

```haskell
data Contest = Rock | Scissors | Paper
data Velocity = MetersPerSecond Float
                | FeetPerSecond Float
```

Example of contest and velocity

```
winner Rock Scissors = ”Player 1”
winner Scissors Paper = ”Player 1”
winner Paper Rock = ”Player 1”
winner Scissors Rock = ”Player 2”
winner Paper Scissors = ”Player 2”
winner Rock Paper = ”Player 2”
winner _ _ = ”Tie”

thrust (FeetPerSecond x) = x / 3.28
thrust (MetersPerSecond x)  = x
```

The Most Fun Datatypes are Recursive

Our Own List Construct

```haskell
data Mylist = Cons Int Mylist
            | Nil
   deriving Show
mklist [] = Nil
mklist (x:xs) = Cons x (mklist xs)
```

We can run it like this:

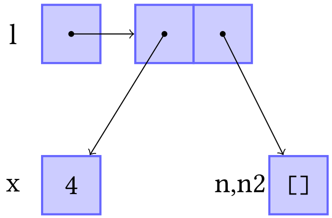
```
*Main> let l1 = mklist [2,3,4]
*Main> l1
Cons 2 (Cons 3 (Cons 4 Nil))
```

1. A recursive type without a recursive case is not really recursive.
2. A recursive type without a base case is dangerous, but using Haskell, it might even make sense.

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.

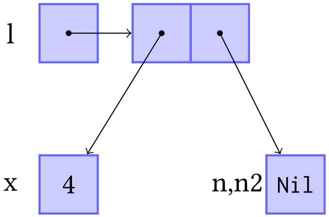
```haskell
x = 4  -- allocates 4
n = []  -- allocates empty list
n2 = n  -- does NOT allocate memory
l = x:n -- A cons cell is allocated, but not the 4 or the empty list
```



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.

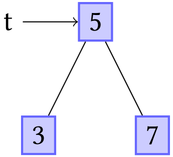
Parametric Polymorphism

```haskell
data Glist a = GCons a (Glist a)
             | GNil
  deriving Show
```

```haskell
x1 = GCons 1 (GCons 2 (GCons 4 GNil))
x2 = GCons "hi" (GCons "there" GNil)
x3 = GCons Nil (GCons (Cons 5 Nil) GNil)
```

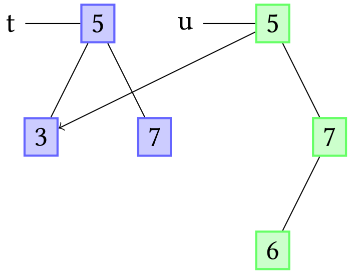
Functional Updating

- 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`



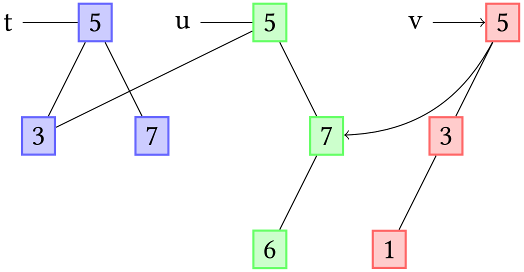
Functional Updating

- 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

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

We can use it in places where we want to return something, but we are not sure that the item exists.

```haskell
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
```