Introduction

Details

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 arithmatic calculation.

Simple Type Definitions

Disjoint Type Syntax

```haskell
data Name = Name [type · · ·] | Name [type · · ·] · · ·
```

A sum type has three components:
- A name to represent the type.
- A set of constructors that each create an instance of that type.
- Constructors and types can both take arguments.

```haskell
data Contest = Rock | Scissors | Paper
data Velocity = MetersPerSecond Float |
FeetPerSecond Float
data List a = Cons a (List a) |
Nil
data Tree a = Node a (Tree a) (Tree a)
```

Example of contest and velocity

```haskell
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 List a = Cons a (List a) |
             Nil

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:

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

**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
          -- call to `insertSorted`
```

**Parameters**

Haskell supports **parametric polymorphism**, like templates in C++ or generics in Java.

**Parametric Polymorphism**

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

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.

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

```
let u = add t 6
let v = add u 1
```

```
t 5
|
3 7
```

```
t 5
|
3 7
```

```
t 5
|
3 7
```

```
t 5
|
3 7
```

```
6
```
The Maybe Type

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

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

```
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

The Either Type

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

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

```
getItem key [] = Left "Key not found"
ggetItem 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"