CptS 355- Programming Language Design

Functional Programming in Haskell
Part 4

Instructor: Jeremy E. Thompson

Spr 2021

Washington State University
World Class. Face to Face.

Haskell Type Synonyms

- We can create new names for existing types (type synonyms) using:

```
type id = type-definition;
```

  - Type synonyms just give another name to an existing type
  - Type synonyms can be used wherever other types are, including inside other type synonyms
  - Examples:

```haskell
type Point = (Float, Float)
type Line = (Point, Point)

-- polymorphic type
type Node a = (a,a)
type Edge a = (Node a, Node a)

node1 = (1,2)::(Node Int)
node2 = (4,8)::(Node Int)
node3 = (6,9)::(Node Int)
edge1 = (node1,node2)::(Edge Int)
edge2 = (node2,node3)::(Edge Int)
```

```haskell
type Graph a = [Edge a]

g = [edge1,edge2]

(x:xs) = g

a = fst x
b = snd x
```

Haskell Data Type Mechanism

- The data type mechanism specifies **new data types** using value constructors
- For example,

Try:
a = Tuesday
b = Tuesday
c = Sunday

:t a
print a
a == b
a == c

```
data Days = Sunday | Monday | Tuesday | Wednesday | Thursday |
            Friday | Saturday
            deriving (Show,Eq)

:t Sunday -- returns Sunday :: Days
```

(optional)
instructs the Haskell compiler to include the new type `Day` into the standard type classes `Show` and `Eq`

  – **The datatype name and data constructors need to start with capital letters**
  – See http://www.haskell.org/onlinereport/derived.html for other possibilities for 'deriving'

Haskell Data Types – Pattern Matching

- Values of enumeration types are often scrutinized by way of pattern matching
- We include the type's data constructors as patterns in the function:
- For example,

```haskell
isWeekday :: Days -> Bool
isWeekday Sunday   = False
isWeekday Saturday = False
isWeekday _        = True

isWeekday Tuesday  -- returns True
```

What if we skip the first line?

  – Alternative implementation of `isWeekday`

```haskell
isWeekday :: Days -> Bool
isWeekday day = not (day `elem` [Saturday, Sunday])
```

    - If we remove Eq from the `deriving` type classes, will the second implementation work?

Haskell Data Types - Parameterized Data Constructors

- The constructors used to define data types may be **parameterized**:

```
data Money = NONE | COIN Int | BILL Int
             deriving (Eq, Show)

x = (COIN 10)
y = (BILL 20)
z = NONE
:t x              -- returns x::Money
```

Standard convention to enclose parameterized types in Parentheses (but not *required*)

- Will the following work?

```
(COIN 5) == (COIN 10)
(COIN 5) == (BILL 5)
(COIN 25) > (COIN 5)
(COIN 25) > (BILL 5)
```

  – How should we change the data type definition to make these work?

Haskell Data Types - Parameterized Data Constructors

- Example function that takes a `Money` value as argument
  - Calculates the amount of a `Money` value in dollars

```
amount :: Fractional p => Money -> p
amount NONE = 0
amount (COIN x) = fromIntegral x/100.0
amount (BILL x) = fromIntegral x

amount (COIN 25) -- returns 0.25
```

The parentheses are necessary—why?

  - We make use of patterns to extract the values of the parameter of data constructs
  - The following **will not work**:

```
amount x =  if x == COIN then ....

amount x =  if typeof(x) == COIN then ....
```

  - How can we write a function that adds two `Money` values?
  - How can we change `amount` function so that it returns a `Money` value (i.e., `BILL or COIN`)?

Haskell Polymorphic Data Types

- Consider the following:

```
data MaybeInt = JustInt Int | NoInt deriving (Show, Eq)
data MaybeStr = JustStr String | NoStr deriving (Show, Eq)
data MaybeDouble = JustDouble Double | NoDouble  deriving (Show, Eq)
```

  - It would be tedious if we had to create such a type and name the constructors for each possible base type
  - Polymorphism will be helpful here

- A user-defined data type may be polymorphic
  - Which means we can have the *type variables* in our user-defined types

Haskell Polymorphic Data types

- Data types (like functions) can be polymorphic.
  - For example:

```
data Maybe a = Just a  | Nothing
               deriving (Show, Eq, Ord)
```

- The values of “Maybe” type either don’t have a value (which is captured by the Nothing construct) or they have a value of some type
  - Examples:

```
a = (Just 10)       -- :t a  returns a :: Num a => Maybe a
b = (Just "ten")    -- :t b  returns b :: Maybe [Char]
c = Nothing         -- :t d  returns d :: Maybe a
d = (Just [1,2,3]) -- :t e  returns e :: Num a => Maybe [a]
```

- The `Maybe` type constructor is pre-defined in Haskell Prelude

Haskell “Maybe” Type

- The `Maybe` type constructor is *pre-defined* in Haskell Prelude

- Why does Haskell introduce the “Maybe” type?
  – In languages like Java or C/C++, object references (or pointers) may be null/undefined
  – In Haskell, as we have seen, all references to data always point to some value
  – However, sometimes you need optional-ness
    - Haskell provides data types in the standard library that allow defining references with *optional* data
  – One common use of `Maybe` type is to handle functions with an optional argument

Functions with “Maybe” Type

1. Head of a list:

```
head' :: [a] -> Maybe a
head' [] = Nothing
head' (x:xs) = (Just x)

a = [[1],[2,3]]
b = []
head a          -- returns [1]
head’ a         -- returns Just [1]
head b          -- returns *** Exception: Prelude.head: empty list
head’ b         -- returns Nothing
```

Functions with “Maybe” Type

2. Last element of a list:

```
last' :: [a] -> Maybe a

last' [] = Nothing
last' [x] = (Just x)
last' (x:xs) = last' xs

a = [[1],[2,3]]
b = []
last’ a          -- returns Just [2,3]
last’ b          -- returns Nothing
```

Functions with "Maybe" Type

3. Tail of a list:

YOUR TURN!

```
tail' :: [a] -> Maybe [a]
tail' [] = Nothing
tail' (x:xs) = (Just xs)

a = "hello"
tail a              -- returns "ello"
tail' a             -- returns Just "ello"
tail' []            -- returns Nothing
```

Functions with “Maybe” Type

3. Add two `Maybe Num` values and return `Maybe Num` value

```
addMaybe :: Num a => Maybe a -> Maybe a -> Maybe a

addMaybe Nothing Nothing = Nothing
addMaybe Nothing (Just v) = (Just v)
addMaybe (Just v) Nothing = (Just v)
addMaybe (Just v1) (Just v2) = (Just (v1+v2))

addMaybe (Just 20) (Just 10)     -- returns Just 30
addMaybe (Just 20)  Nothing      -- returns Just 20
```

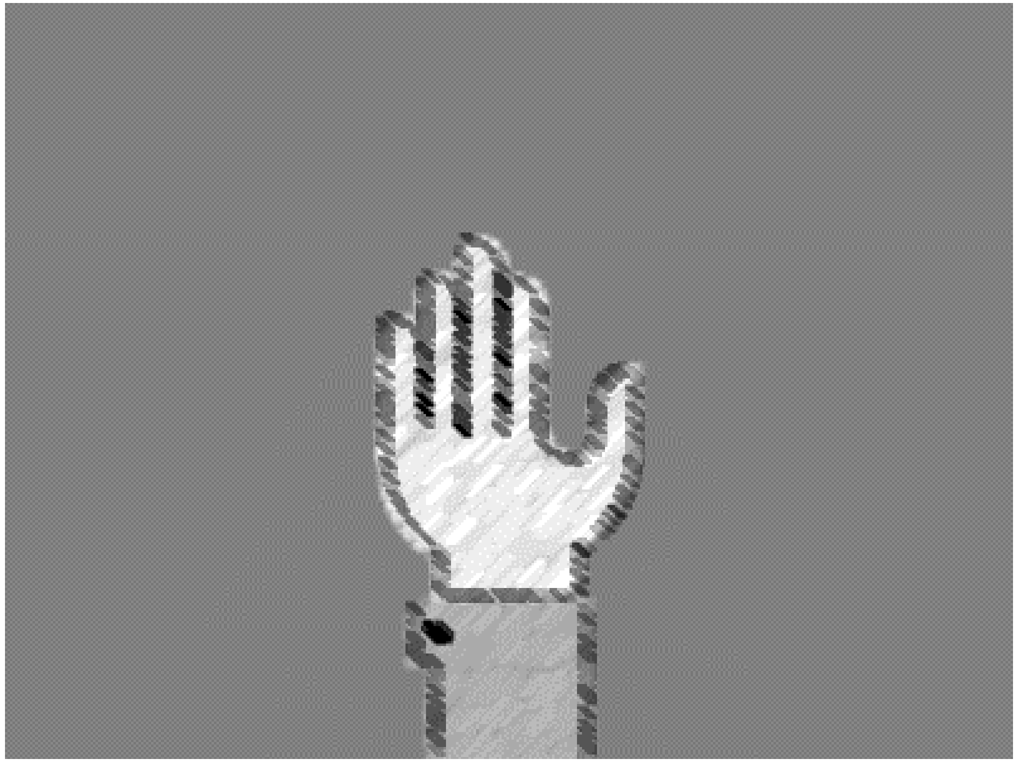
4. Add two `Maybe Num` values and return `Num` value

```
addMaybe :: Num a => Maybe a -> Maybe a -> a

addMaybe Nothing Nothing = 0
addMaybe Nothing (Just v) = v
addMaybe (Just v) Nothing = v
addMaybe (Just v1) (Just v2) = v1+v2

addMaybe (Just 20) (Just 10)     -- returns 30
addMaybe (Just 20)  Nothing      -- returns 20
```

Questions?



Data types may be recursive

- Recursive datatypes allow linked structures without explicit pointers
  - For example:

```
data MyList a = EMPTY | CONS a (MyList a)
```

- `Mylist` is a data type which is recursive
  - We will see later that these recursive data types can naturally be processed using recursive functions

- The type we just defined here is essentially the *same type* as the Haskell list
  - Fundamentally there is no difference—they both serve the same purpose—but the compiler recognize them as separate types

```
:t (:)
(:) :: a -> [a] -> [a]

:t CONS
CONS :: a -> MyList a -> MyList a
```

Recursive Types

- **Tree data type:**
  - A tree consists of a <u>root</u> (maybe all by itself) and *optionally* some <u>children</u> which are also trees
  - Assume we create a *binary* `IntTree` where <u>`Int` values are stored in the leaves and no values will be stored in the interior nodes</u>

```
data IntTree = Leaf Int | Node (IntTree) (IntTree)
               deriving (Show, Eq)

l1 = Leaf 3
l2 = Leaf 4
tree1 = Node l1 l2

tree2 = Node (Node (Leaf 5) (Leaf 6)) (Leaf 4)
```

Note: `deriving (Show)` is required to print trees
`deriving (Eq)` is required to compare trees

Recursive Types

- **Polymorphic Tree Data Type:**

Can specify type 'a' here to allow internal nodes to store values

```
data Tree a = LEAF a | NODE (Tree a) (Tree a)
              deriving (Show, Eq)

l1' = (LEAF "three");
l2' = (LEAF "four");
tree3 = NODE l1' l2'

tree4 = NODE (NODE (LEAF "one") (LEAF "two"))
             (NODE (LEAF "three") (LEAF "four"))
```

Now we can create trees of various types

Recursive Types

- **Alternative Tree Types:**

1. Polymorphic *ternary* tree, *values are stored <u>only</u> in the leaves*

```
data TriTree a = TLEAF a | TNODE (TriTree a) (TriTree a) (TriTree a)
                 deriving (Show, Eq)

ttree = TNODE (TLEAF 9)
              (TNODE (TLEAF 8) (TLEAF (-1)) (TLEAF 7))
              (TNODE (TLEAF 3) (TLEAF 4) (TNODE (TLEAF 1)
                                                (TLEAF (-8))
                                                (TLEAF 5) ))
```

2. Polymorphic *binary* tree, *values are stored <u>both</u> in leaves and the interior nodes*

Now we specify a value in the interior nodes

```
data BinTree a = BLEAF a | BNODE a (BinTree a) (BinTree a)

btree = BNODE "one" (BLEAF "two")
                    (BNODE "three" (BLEAF "four") (BLEAF "five") )
```

Functions to Process Recursive Types

- We can use recursive functions to process the values in recursive tree types

```
data Tree a = LEAF a | NODE (Tree a) (Tree a)
              deriving (Show, Eq)
```

- There is a strong connection between the structure of the recursion and the structure of the recursive data type
- Example:
  - Function to count the number of leaves in a tree:

Give the definition of nNodes (number of nodes)

```
nLeaves :: Num p => Tree a -> p
nLeaves (LEAF _) = 1
nLeaves (NODE t1 t2) = (nLeaves t1) + (nLeaves t2)
```

```
myTree = NODE (NODE (LEAF "one") (LEAF "two")) (NODE (LEAF "three") (LEAF "four"))
nLeaves myTree --returns 4
```

Functions to Process Recursive Types

```
data Tree a = LEAF a | NODE (Tree a) (Tree a)
              deriving (Show, Eq)
```

- Copy tree:

```
copyTree :: Tree a -> Tree a
copyTree (LEAF x) = LEAF x
copyTree (NODE t1 t2) = NODE (copyTree t1) (copyTree t2)
```

```
myTree = NODE (NODE (LEAF "one") (LEAF "two")) (NODE (LEAF "three") (LEAF "four"))
copyTree myTree
-- returns NODE (NODE (LEAF "one") (LEAF "two")) (NODE (LEAF "three") (LEAF "four"))
```

Change binary tree to ternary; rewrite it

Functions to Process Recursive Types

```
data Tree a = LEAF a | NODE (Tree a) (Tree a)
              deriving (Show, Eq)
```

- Tree Map:

```
treeMap :: (t -> a) -> Tree t -> Tree a
treeMap op (LEAF x) = LEAF (op x)
treeMap op (NODE t1 t2) = NODE (treeMap op t1) (treeMap op t2)
```

```
myTree = NODE (NODE (LEAF "one") (LEAF "two")) (NODE (LEAF "three") (LEAF "four"))
strUpper s = map toUpper s   --must import Data.Char
treeMap strUpper myTree
--returns NODE (NODE (LEAF "ONE") (LEAF "TWO")) (NODE (LEAF "THREE") (LEAF "FOUR"))
```

Change tree to store values in nodes, rewrite it

Functions to Process Recursive Types

```
data TriTree a = TriLEAF a |
                 TriNODE a (TriTree a) (TriTree a) (TriTree a)
                 deriving (Show, Eq)
```

Try coding postOrderTri

Preorder Traversal:

```
preOrderTri :: TriTree a -> [a]
preOrderTri (TriLEAF x) = [x]
preOrderTri (TriNODE x t1 t2 t3) = [x] ++ (preOrderTri t1) ++
                                   (preOrderTri t2) ++ (preOrderTri t3)
```

```
myTriTree = TriNODE 0 (TriNODE 9 (TriLEAF 1) (TriLEAF 2) (TriLEAF 6) )
                      (TriNODE 8 (TriLEAF 3) (TriLEAF 4) (TriLEAF 7) )
                      (TriLEAF 5)
preOrderTri myTriTree

--returns [0,9,1,2,6,8,3,4,7,5]
```

Questions?