COURSE NAME: CS558

COURSE DATE: 9/13/18

Lecture

Last time, we looked at mostly trees of polymorphic types:

data Tree a = Leafa

| Node (Tree a) (Tree a)

treeMap :: (a ->b) -> (Tree a) -> (Tree b)

treeMap f (Leafx) = Leaf (f x)

treeMap f (Node t1 t2) = Node (treeMap f t1) (treeMap f t2)

We also looked at fold, which is different than the one for tree:

treeFold (a -> b) -> (b-> b ->b) -> (Tree a) -> b

Start with a -> b, which returns the value of type b that corresponds.

a -> b corresponds to the leaf constructor and b -> b -> b corresponds to the node.

treeFold fLeaf fNode (Leaf x) = fLeaf x

treeFold fLeaf fNode (Node t1 t2) = fNode (treeFold fLeaf fNode t1) (treeFold fLeaf fNode t2)

Proving functions:

First prove the base case (prove for all leaves) and then prove the inductive case (using both subtrees).

Haskell Types

You have to provide at least 1 type instructor, for instance if you want to know if the data is a leaf or a node:

1. “data” keyword -> data Tree a = Leaf a

| Node (Tee a) (Tree a)

1. Type synonyms 🡪 string type is really just a synonym for a list of characters.

type String = [Char ]

typeName = String

x = “John Smith” :: Name

y = “Hello” :: String

x = = y 🡪 this is ok because type actually matters!

1. Record syntax 🡪 this is like record data, but you can provide names for the fields (arguments to type constructors)

What if we want to represent shapes with x, y coordinates?

data ShapeCoords = Circle Double Double Double

| Square Double Double Double

In record syntax

data ShapeCoords = Circle { xCoord : : Double,

yCoord : : Double,

zCoord : : Double }

| Square { xCoord : : Double,

yCoord : : Double,

zCoord : : Double }

unit CircleAtOrigin = Circle { xCoord = 0 – 0,

yCoord = 0 – 0,

radius = 1-0}

OR:

unit CircleAtOrigin = Circle { radius = 1.0,

xcoord = 0.0,

ycoord = 0.0

\*Ordering doesn’t matter if you label the arguments!

OR:

unit CircleAtOrigin = Circle 0 . 0 0 . 0 1 . 0

\*If no labels are provided, the ordering from the original definition is assured.

Automatically get “assessor” functions!

1. newType

Allows you to wrap an existing type as if it were a new type.

If you have a type in a type class like show, and you wanted to derive a new default behavior for that type class, you can use newType.

Triples

(1, 2) : : (Int, Int)

(“Hello”, 7, [10, 2.0]) : : (String, Int, [Float])

sumPair : : (Int, Int) -> Int

sumPair (x, y) = x + y

fst (a, b) ->a snd : : (a, b) -> b

fst (x, - ) = x snd ( -, y) = y

fst (1, 2, 3) 🡪 this would give a type error!

Modules

Organizes code into name space.

Prelude – default module, head, tail, (+), ( : ), (+ +)

Import modules to access their contacts, e.g. Data.List

There’s a function from Data.List called nub: Eq a -> [a] -> [a]

It removes duplicates from List! You need to import items from the name space.

1. Import Data.List

Everything in Data.List module is added to the correct namespace.

nub [1, 1, 2, 2, 3]

1. Import qualified Data.List

You have to give full names as in Data.List.nub [1, 1, 2, 2, 3]

1. You can give it a shortened name. Import qualified Data.List as L

L. nub [1, 1, 2, 2, 3]

Create a module Foo

Make a file Foo.hs containing module Foo whose x = 2 and f y = y +x

Can then do import qualified Foo

(Foo) x

{Foo) y 2 etc….

I/O in Haskell

Haskell is pure language, so no side effects and lazy evaluation. If the expression is not needed, it’s never evaluated.

In most cases, it doesn’t matter where your code is evaluated.

But I/O does have side effects (such as writing to a file). So we use a type system that lets us specify and test properties of a program without running it. Use types to “encapsulate” the effects to I/O.

Type IO a for I/O actions that produce a value of type a.

Example reading and printing a file:

import System.Environment

main : : IO ( ) 🡪 the empty brackets are a type “unit”

main = do test <- readFile “in.txt”

putStrLn text

To run this program: runhaskell myProgram.hs

Produces a binary called myProgram

There are two types of comments in Haskell (in case you want them in your submission)

* - this is a single line comment.

{ - this is a multiline comment - }

##end notes##