More Haskell

Nathan Whitehead
nwhitehe@ucsc.edu
Baskin Engineering 259A
Office hours: Tues 10-11 am 4-5 pm, Thur 4-5 pm
Functions are Data

Perhaps the most powerful capability of Haskell:

- functions are just another form of data
- can be passed around
- first-class values
Functions in Tuples

plus1 :: Int -> Int
plus1 x = x + 1

minus1 :: Int -> Int
minus1 x = x - 1

funcpair = (plus1, minus1)

:type funcpair
funcpair :: (Int -> Int, Int -> Int)
Functions in Lists

plus1 :: Int -> Int
plus1 x = x + 1

minus1 :: Int -> Int
minus1 x = x - 1

funclist = [plus1, minus1, plus1]

:type funclist
funclist :: [Int -> Int]
applyTwice f x = f (f x)

applyTwice plus1 5
7

applyTwice minus1 5
3

:type applyTwice
applyTwice :: (t -> t) -> t -> t
\[
\text{plusn :: Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]
\[
\text{plusn } n = f
\]
\[
\text{where } f \ x = x + n
\]

\[
\text{plus10} = \text{plusn} \ 10
\]

\[
\text{plus10} \ 12
\]

\[
22
\]
Anonymous Functions

When thinking of names is just too much work...

\(\lambda x \rightarrow x + 1\)

\((\lambda x \rightarrow x + 1)\ 100\)

101

\(\text{map}\ (\lambda x \rightarrow x + 1)\ [1,\ 2,\ 3]\)

[2,3,4]
Infix Operators

Many operators are infix to match math usage

3 + 2
5

(+) 3 2
5

map ((+) 2) [1, 2, 3]
[3,4,5]
Infix Operators

Can make syntax more flexible

\[(\mid>) \; x \; f = f \; x\]

0 \mid> (+) 2 \mid> (+) 3
5

2 `plus` 3
[3,4,5]

map (10 `minus`) [1, 2, 3]
[9,8,7]
You can define your own infix operators, declare associativity and precedence.

```haskell
infixl 5 <=>
(<=>) :: Bool -> Bool -> Bool
x <=> y = (x == y)

True <=> False
False
```
Simple Debugging

It can be nice to see what you're doing...

data CoordT = Coord Double Double

3.0
3.0

(3.0, 5.0)
(3.0,5.0)

(Coord 3.0 5.0)

No instance for (Show CoordT)...

It can be nice to see what you're doing...

data CoordT = Coord Double Double deriving (Show)

3.0
3.0

(3.0, 5.0)
(3.0,5.0)

(Coord 3.0 5.0)
Coord 3.0 5.0
data CoordT = Coord Double Double
  deriving (Show, Eq, Ord, Read)

Coord 3 5 < Coord 4 6
True

Coord 3 5 == Coord 4 6
False
Recursive Types

data IList = INil | ICons Int IList
    deriving (Show)

ICons 3 (ICons 5 INil)
ICons 3 (ICons 5 INil)

:t ICons
ICons :: Int -> IList -> IList
data List a = Nil | Cons a (List a) 
  deriving (Show)

Cons 3 (Cons 5 Nil)
Cons 3 (Cons 5 Nil)

:t Cons
Cons :: a -> List a -> List a
Parametric types add a notion of 'kind': the type of types.

`:kind Int
Int :: *
:`kind (Int -> Int)
(Int -> Int) :: *
`:info List
data List a = Nil | Cons a (List a)
`:kind List
List :: * -> *
`:kind (List Int)
(List Int) :: *
data Either a b = Left a | Right b

:kind Either
Either :: * -> * -> *
:kind Either String Int
Either String Int : *

data MetaData a b = RedData (a b)
   | BlackData (a b)

:kind MetaData
MetaData :: (* -> *) -> * -> *

Typeclasses are like interfaces, define behavior that \textit{types} must have.

\begin{align*}
(+):& \text{ Num } a \Rightarrow a \rightarrow a \rightarrow a \\
\text{elem}:& \text{ Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{ Bool} \\
\text{sort}:& \text{ Ord } a \Rightarrow [a] \rightarrow [a]
\end{align*}

\text{class Eq } a \text{ where} \\
\begin{align*}
(==):& \text{ a } \rightarrow \text{ a } \rightarrow \text{ Bool} \\
(\neq):& \text{ a } \rightarrow \text{ a } \rightarrow \text{ Bool} \\
x == y &= \text{ not } (x \neq y) \\
x \neq y &= \text{ not } (x == y)
\end{align*}
Typeclasses

Typeclasses are like interfaces, define behavior that *types* must have.

```haskell
data CoordT = Coord Double Double

Coord 4 0 == Coord 4 3
    No instance for (Eq CoordT)

instance Eq CoordT where
    (Coord x y == Coord x1 y1) = (x == x1)

Coord 4 0 == Coord 4 3
    True
```
Defining a Typeclass

Let's make a typeclass for measuring “size” of data structures.

class Measurable a where
  measure :: a -> Int

data Tree a = Empty
  | Node a (Tree a) (Tree a)

instance Measurable (Tree a) where
  measure Empty = 0
  measure (Node x l r) =
    1 + (measure l) + (measure r)
Type inference detects when we need our typeclass.

\[
\text{measure2 } x = 2 * \text{measure } x
\]

\[
:t \text{ measure2} \\
\text{measure2 :: Measurable } a \Rightarrow a \rightarrow \text{Int}
\]
Using Our Typeclass

We can instantiate our typeclass for parametric types.

```
instance Measurable [a] where
  measure [] = 0
  measure (x:xs) =
    measure x + measure xs
```

No instance for (Measurable a)...

We can instantiate our typeclass for parametric types.

```haskell
instance (Measurable a) => Measurable [a] where
  measure [] = 0
  measure (x:xs) = measure x + measure xs
```
Type classes are somewhat reminiscent of object classes...

- Type classes are *sets of types* that have the proper operations defined
- Functions inside type classes are *type-class polymorphic*
- Compiler is using compile time *type inference* to decide which implementation to use (not vtables)
- Dispatch can be based on any parameter types of type class
IO in a Functional Language

```haskell
putStrLn "Hello"
Hello

:t putStrLn
putStrLn :: String -> IO ()

:t putStrLn "Hello"
putStrLn "Hello" :: IO ()
```
main = do
    putStrLn "What does the fox say?"
    fox <- getLine
    putStrLn $ "Yiyiyiyiyaya " ++ fox

:type getLine
getLine :: IO String
:type main
main :: IO ()
IO actions are values, can be treated as such

actions = [putStrLn "Hello ", putStrLn "world"]

:t actions
actions :: [IO ()]

seqz :: [IO ()] -> IO ()
seqz [] = return ()
seqz (x:xs) = do x
               seqz xs
do is syntactic sugar

seqz :: [IO ()] -> IO ()
seqz [] = return ()
seqz (x:xs) = x >>= seqz xs

(>>) :: IO a -> IO b -> IO b

do
  x <- getLine
  putStrLn x

g getline >>= (\x -> putStrLn x)

(>>=) :: IO a -> (a -> IO b) -> IO b
IO is an example of Monad

type IO a = Reality -> (Reality, a)

class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a

:tl (>>=)
(>>=) :: Monad m => m a -> m b -> m b
 :tl seqz
seqz :: Monad m => [m a] -> m b
Don't need a deep understanding of monads to use them
Laziness

Evaluation is *non-strict*: expressions are not reduced until/unless needed

- What do we get?
  - Avoid repeated passes over lists
  - Avoids infinite loops
  - Lets us use infinite values (not all at once...), streams
  - Short-circuit boolean evaluation is simple example
Infinite List

inc = 1 : map (+1) inc

take 10 inc
[1,2,3,4,5,6,7,8,9,10]

:t inc
inc :: [Int]

inc !! 10000
10001
Infinite List

\[
fibs = 1 : 1 : \text{zipWith (+) fibs (tail fibs)}
\]

\[
\text{take 10 fibs}
[1,1,2,3,5,8,13,21,34,55]
\]

\[
: \text{t fibs}
\]

\[
fibs :: [\text{Int}]
\]

\[
: \text{t zipWith}
\]

\[
\text{zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]}
\]
Infinite Recursion

loop = loop

loop

:t loop
loop :: t

looplist = [5, loop, 100, loop]
looplist !! 2
100
looplist !! 1
class Functor f where
    fmap :: (a -> b) -> f a -> f b

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

instance Functor List where
    fmap = ???
class Functor f where
    fmap :: (a -> b) -> f a -> f b

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

map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)

instance Functor List where
    fmap = map
class Functor f where
    fmap :: (a -> b) -> f a -> f b

data Maybe a = Just a | Nothing

instance Functor Maybe where
    fmap f ... = ???
class Functor f where
    fmap :: (a -> b) -> f a -> f b

data Maybe a = Just a | Nothing

instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
class Functor f where
  fmap :: (a -> b) -> f a -> f b

data Tree a = Empty
  | Node a (Tree a) (Tree a)

instance Functor Tree where
  fmap f Empty = Empty
  fmap f (Node a l r) =
    Node (f a) (fmap f l) (fmap f r)
Towers of Hanoi

• An example combining IO and recursion
Towers of Hanoi

Object: to move disks from post 1 to 3

Never put a bigger disk on a smaller one
Towers of Hanoi

To move k stack from post x to y using z:
  Recursively move k-1 stack from x to z
  Move disk from x to y
  Recursively move k-1 stack from z to y
Towers of Hanoi

hanoi n a b c
-- Move n disks from a to b using c

hanoi :: Int -> String -> String -> String -> IO ()
  String -> IO ()

hanoi 2 "a" "b" "c"
Move a to c
Move a to b
Move c to b
Towers of Hanoi

hanoi 1 a b c = do
  putStrLn $ "Move " ++ a ++ " to " ++ b

hanoi n a b c = do
  ???
hanoi 1 a b c = do
    putStrLn $
        "Move " ++ a ++ " to " ++ b

hanoi n a b c = do
    hanoi (n - 1) a c b
    putStrLn $
        "Move " ++ a ++ " to " ++ b
    hanoi (n - 1) c b a
End