CMPT 383

Haskell Types

Anders Miltner
Lecture Goals

• Describe Haskell types

• Describe the relationship between terms and structural typing

• Go into type and term syntax
Basic Types

• Tuples (products)
• Disjoint Unions (sums or coproducts)
• Polymorphism (foralls)
• Arrows (functions)
• Tuples!
• Basic building block in Haskell
• Fixed length, with fixed types for each element of the tuple
Building tuples

tuples_builder :: a -> b -> (a,b)
tuples_builder x y = (x,y)
Extracting Information from tuples

```plaintext
let tuple = (x,y) in
let x' = fst (x,y) in
let y' = snd (x,y) in
let (x'',y'') = tuple in
```
Can build arbitrary-length tuples

```haskell
big_tuple_builder :: a -> b -> c -> (a,b,c)
tuples_builder x y z = (x,y,z)

let tuple = (x,y,z) in
--let x' = fst (x,y) in
--let y' = snd (x,y) in
let (x'',y'',z'') = tuple in
```
Data Types / Sum Types

- This is where functional programming really starts to shine
- Some languages have *very recently* started adopting them
- Think of them like enums with extra information
Data Types by Example

data Bool =
    True
    | False
Data Types by Example

data Bool =
  True
| False

Looks like they are just enums right?
Data Types by Example

```haskell
data MaybeInt =
  Nothing
| Just Int

divide :: Int -> Int -> MaybeInt
divide i j =
  if j == 0 then
    Nothing
  else
    Just (i/j)
```
Data Types by Example

show_result :: MaybeInt -> String
show_result Nothing = "No Int"
show_result Just i = "Int of " ++ (show i)

Just like lists!
Lists as Data Types?

data List =
   Nil
| Cons (int,List)
data RoseTree =
  Leaf
  | Node (Int,List)

data List =
  Nil
  | Cons (RoseTree,List)
data Type =
  Base Int
  | Product ([Type])
  | Sum ([(String, Type)])
  | Arrow (Type, Type)
  | Poly (String, Type)
  | Variable String
Records

- Sometimes it is nice to have names the parts of large tuples
- You can name the constituent components
Record Example

data Configuration = Configuration
{ userName :: String,
  isGuest :: Bool
}

x :: Configuration
x = Configuration { userName = "TestUser",
  isGuest = True }

testUserName :: String
testUserName = userName x
Polymorphic Types

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

data Tree a =
  Leaf
  | Node (Tree a,a,Tree a)
data RoseTree a =
    Leaf
    | Node (a,List (RoseTree a))

data List a
    Nil
    | Cons (a,List a)
Polymorphic Types

data List2 a b =
  Nil
  | Cons1 (a, List2 a b)
  | Cons2 (b, List2 a b)

type IntOrBoolList = List2 Int Bool

myList :: IntOrBoolList
myList = Cons1 (1, Cons2 (True, Nil))
Polymorphic Types

```haskell
data List2 a b =  
  Nil  
  | Cons1 (a,List2 a b)  
  | Cons2 (b,List2 a b)

type IntOrSomethingList a = List2 Int a

myList :: IntOrSomethingList Bool
myList = Cons1 (1, Cons2 (True, Nil))
```
Polymorphic Types

data SwapList a b =
    Nil
  | Cons (a, SwapList a b)
  | ConsSwap (a, SwapList b a)

swapping :: SwapList Int Bool
swapping =
    Cons (0, ConsSwap (1, Cons (True, Nil)))
High Level Intuition on Polymorphic Instantiation

• Can you replace the type variables to find a single coherent type?
• If so, then everything is well-typed
• This process of finding those variable replacements is called unification
Unification Example 1

• x1 :: Int

• x2 :: Int

• x3 :: Int

• createTriple :: a -> b -> b -> (a,b,b)
  createTriple x y z = (x,y,z)

• :t createTriple x1 x2

• Int -> (Int,Int,Int), Well-Typed by replacing a with Int, b with Int!
Unification Example 2

- $x_1 :: \text{Float}$
- $x_2 :: \text{Int}$
- $x_3 :: \text{Int}$

- $\text{createTriple :: a -> a -> b -> (a,a,b)}$
  $\text{createTriple} \ x \ y \ z = (x,y,z)$

- :t $\text{createTriple} \ x_1 \ x_2 \ x_3$

- Not well typed, there is no instantiation of a and b consistent with $(\text{Float}, \text{Int}, \text{Int})$
Unification Example 3

• \( x_1x_3 :: (\text{List } c, \text{List } c) \)

• \( x_2 :: \text{Int} \)

• \( \text{createTriple} :: a \rightarrow a \rightarrow b \rightarrow (a,a,b) \)
  \[ \text{createTriple } x \ y \ z = (x,y,z) \]

• \( :t \text{createTriple } (\text{fst } x_1x_3 , x_2 , \text{snd } x_1x_3) \)

• Not well typed! There is no instantiation of \( a \) and \( b \) consistent with \( (\text{List } c, \text{Int}, \text{List } c) \)
Unification Example 4

• $x_1 x_3 :: (\text{List } a, \text{List } a)$

• $x_2 :: \text{List } b$

• $\text{createTriple} :: c \to c \to d \to (c, c, d)$
  \[
  \text{createTriple } x \ y \ z = (x, y, z)
  \]

• $:t \text{createTriple} (\text{fst } x_1 x_3, x_2, \text{snd } x_1 x_3)$

• Well-typed! With type $(\text{List } a, \text{List } a, \text{List } a)$
  This is due to the fact that the $b$ in $x_2$ can be replaced by $a$, and both $c$
  and $d$ are replaced by $\text{List } a$ as well
Arrow Types

• T1 -> T2

• You’ve probably already seen this…
  • But it’s about to get way harder

• Arrow is a binary operation

• f :: T1 -> T2 means that:
  • if f is given an element of T1 as input, then T2 will be output
So what does $T_1 \rightarrow T_2 \rightarrow T_3\ldots$ mean?

- The $\rightarrow$ type is right associative
  - $T_1 \rightarrow T_2 \rightarrow T_3$ means $T_1 \rightarrow (T_2 \rightarrow T_3)$
  - So, given a type in $T_1$, produce a value of type $T_2 \rightarrow T_3$
    - Which means, when it is given an element in $T_2$, it will produce something in $T_3$
What do other languages do?

• All functions have type \((T_1, \ldots, T_n) \rightarrow T\)

• What does this mean?
  • Harder to make partial application

• Are they equivalent?
  • Actually yes! In any reasonable & classical (aka non-quantum) language

• Look up “cartesian closed categories” if you’re interested!
Partial Application Uses

• “Cleaner” function definitions

• \(3\text{mod}k \colon \text{Int} \to \text{Int}\)
  \[3\text{mod}k \, k = \text{mod} \, 3 \, k\]

• \(3\text{mod}k \colon \text{Int} \to \text{Int}\)
  \[3\text{mod}k = \text{mod} \, 3\]

• Can be useful in higher order function use (see next slide)
What does \((T_1 \rightarrow T_2) \rightarrow T_3\) mean?

- It’s definitely not the same as \(T_1 \rightarrow T_2 \rightarrow T_3 = T_1 \rightarrow (T_2 \rightarrow T_3)\)
- Given an element of type \(T_1 \rightarrow T_2\), produce an element \(T_3\)
- This is called a *higher-order function*
- These are INCREDIBLY useful beyond the scope of this class and functional programming
  - If you had to learn one thing from this class, learn this
Simple Example: Flip

- \( \text{flip} :: (a \to b \to c) \to b \to a \to c \)
  - \( \text{flip } f \ x \ y = f \ y \ x \)

- \( \text{flip mod } 3 \ 6 = 0 \)
Higher Order Functions Example 1

• map :: (n -> m) -> [n] -> [m]

• map (+1) [1,2,3] = [2,3,4]
  map (show) [1,2,3] = ["1","2","3"]
Higher Order Functions Example 2

• filter :: (n -> bool) -> [n] -> [n]

• filter (isEven) [1,2,3] = [2]
  filter (isOdd) [1,2,3] = [1,3]
Higher Order Functions Example 3

• \texttt{foldr :: (n \rightarrow m \rightarrow m) \rightarrow [n] \rightarrow m \rightarrow m}

• \texttt{foldr (+) [1,2,3] 0 = 6}
  \texttt{foldr (++) [[1],[2,3]] [] = [1,2,3]}

foldr :: (n -> m -> m) -> [n] -> m -> m
foldr f [] init = init
foldr f (h:t) init =
  f h (foldr f t init)
Higher Order Functions Example 4

- \( \text{filterMap} :: (n \rightarrow \text{Maybe } m) \rightarrow [n] \rightarrow [m] \)

- \( \text{filterMap } (\text{toInt}) \ [1.0,\ 2.1,\ 3.0] = [1,3] \)
What's In a Name?

- If we want to be passing functions around, it’s going to be pretty annoying to always have to write our functions as “where”s or in the top-level

- I might want to just quickly write out a small function, and not have to go through the overhead of naming it

- These are called anonymous functions
  - This is done with a lambda — the basis of the lambda calculus
Lambda Functions

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

```haskell
fib :: Int -> Int
fib = \x ->
    case x of
    0 -> 0
    1 -> 1
    n -> fib (n-1) + fib (n-2)
```
HOFs can be Anywhere!

```haskell
data Function =
  Constant Float
| Linear (Float,Float)
| Quadratic (Float,Float,Float)
| Arbitrary (Float -> Float)

runFunc :: Function -> Float -> Float
runFunc (Constant i) _ = i
runFunc (Linear (i,j)) x = i*x + j
runFunc (Quadratic (a,b,c)) x = a*x*x + b*x + c
runFunc (Arbitrary f) x = f x
```

```haskell
data Precision = Exact | Approx

data Quantity =
  Finite of (Float list)
| AllReals

zeros :: Function -> (Precision,Quantity)
zeros (Constant i) 0 = (Exact, AllReals)
zeros (Constant i) _ = (Exact, Finite [])
zeros (Linear (i,j)) x = (Exact, Finite …)
zeros (Quadratic (a,b,c)) x = (Exact, Finite …)
zeros (Arbitrary f) x = (Approx,…)
```
Objects as Functional Types

data Dict k v = Set
  { isKey :: k -> bool
  , isValue :: v -> bool
  , lookup :: k -> Maybe v
  }
“Quiz” 1

\[ \text{foo :: (n -> n) -> Int -> n -> n} \]

What does “foo” take as inputs?

foo takes a function from n to n
and an integer
and an element of n
and will product an element of n
Describe the “Foo” data type

Foo is a tree-like data type
The “X” constructor is like a Leaf / end of the tree
The “Y” constructor takes in an element of type a, and a Foo of Ints, and Foo of Bools
This means that the resulting tree will have an element of type a at its root, and all left subtrees will have Ints, and right subtrees Bools
“Quiz” 3

\[ x = ((a, b), c), (d, f) \]

How would you extract \( b \) from \( x \)?

\[ \text{snd (fst (fst } x)) \]
More Function Fun!

- There's some nice built-in functions for manipulating Haskell functions

\[
(\cdot) :: (b \to c) \to (a \to b) \to a \to c
\]
\[
(\cdot) \ g \ f \ x = g \ (f \ x)
\]

\[
\text{maxString :: Int List} \to \text{String}
\]
\[
\text{maxString} = \text{show} \ . \ \text{maximum}
\]

Writing functions this way is called point-free style
More Function Fun!

- There's some nice built-in functions for manipulating Haskell functions

```haskell
($) :: (a -> b) -> a -> b
($) f x = f x

maxString :: Int List -> String
maxString x = show $ maximum x
```