Haskell user defined types

- `data Temp = Cold|Hot|Warm`
  - deriving (Show, Eq, Ord, Enum)
  - -- to enable printing to screen
  - -- comparing for equality
  - -- comparison of order such as `x < Warm`
  - -- use in enumerations such as `[Cold .. Warm]`

  A term an enumerated type

  `Cold` and `Hot` are termed a constructor of type `Temp`. Constructors must begin with a capital letter.

```haskell
data Temp = Hot|Cold|Warm
  deriving (Show, Eq)

data Season = Spring|Summer|Fall|Winter
  deriving (Show, Eq)
weather Winter = Cold
weather Summer = Hot
weather _ = Warm
:
weatherSeason -> Temp
```

- `data Shape = Circle Float | Rectangle Float Float`
  - deriving (Eq, Ord, Show)
    - Termed a composite type

- `data List = Empty | Cons Int List`
  - Termed a recursive type

```haskell
data List a = Empty | Cons a (List a)
data Tree a = Null | Node a (Tree a) (Tree a)
```

- `Termed parametric types (as uses type variables - polymorphic)`
Haskell user defined types

```haskell```
data Student = USU String Float
  deriving (Show)
```

Suppose you have a list of students (using the above type definition), create a list of students with a GPA > 3.0

Note: String is the same as [Char]

What is the TYPE of your function?

```haskell```
data Student = USU String Float
  deriving (Show)

myclass = [USU "Mike" 3.7, USU "Steve" 3.9, USU "Fred" 2.9, USU "Joe" 1.5]

gpa xs = [(USU n g)| (USU n g) <- xs, g > 3.0]
:t gpa
  [Student] -> [Student]
Write a linked list insertion

```haskell
data List = Nil | Node Int List
    deriving (Show)

Make a linked list of Nodes from an int list
makeList [1,2,3]
yields Node 1 (Node 2 (Node 3 Nil))
Write ordered linked list insertion.
insert x List
insert 5 makeList [1,3,4,6]
yields Node 1 (Node 3 (Node 4 (Node 5 (Node 6 Nil)))))
```

data List = Nil | Node Int List
    deriving (Show)

```haskell
makeList :: [Int] -> List
makeList [] = Nil
makeList (x:xs) = (Node x (makeList xs))
addList x rest = (Node x rest)

addOrdered x Nil = Node x Nil
addOrdered x (Node y rest) = if x < y then
    (Node x (Node y rest)) else Node y
    (addOrdered x rest)
```
Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

\[
\text{abs } n
\]
\[
\text{otherwise} = -n
\]

\[
\text{sign } x
\]
\[
\text{otherwise} = \text{False}
\]

_ is wildcard. Patterns are matched in order. For example, the following definition always returns False:

\[
_ \text{&& } _ = \text{False}
\]

Patterns may not repeat variables. For example, the following definition gives an error:

\[
b \text{&& } b = b
\]

\[
_ \text{&& } _ = \text{False}
\]
Nested generators

\[ \{ (x,y) \mid y \leftarrow [4,5], x \leftarrow [1,2,3] \} \]

\[ (1,4), (2,4), (3,4), (1,5), (2,5), (3,5) \]

\( x \leftarrow [1,2,3] \) is the last generator, so the value of the \( x \) component of each pair changes most frequently.

Dependent Generators

Later generators can depend on the variables that are introduced by earlier generators.

\[ \{ (x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3] \} \]

The list \([ (1,1),(1,2),(1,3),(2,2),(2,3),(3,3) ]\) of all pairs of numbers \((x,y)\) such that \(x,y\) are elements of the list \([1..3]\) and \(y \geq x\).
includes x (y:ys) = if x==y then True else includes x ys

:t includes

a->[a] -> Bool

It clearly means input a, [a] -> output Bool, BUT it doesn’t say it that way.

Explain the type of map

- :t map
- (a-> b)->[a]->[b]

Analogy

Suppose you had a procedure to check the balance on a specific account. Would it be helpful to have a specific version of the procedure to check your bank account at CVB?

getBalance (435224332) vs getBalanceCVB

Suppose you know how to call anyone, would it be helpful to have a specific version of your call routine to call Sue?

phone (435-797-2022) vs phoneSue
Curried Functions

\texttt{mult} :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}

\texttt{mult} \ x \ y = x * y

- We could read this as a function taking two \texttt{Int} arguments and returning a third. OR-- a function taking a single \texttt{Int} argument and returning a function from \texttt{Int} \to \texttt{Int} as a result.
- With currying: the \texttt{mult} function can take either one or two parameters; if you give it one, it returns a function where the first argument is always fixed.
- Assume \texttt{doit} takes a function and a list and applies the function to each element of the list.
- Ex: \texttt{doit (mult 3) [1 .. 3]}
  Yields \([3,6,9]\)

So...

- \texttt{take} \ n \ \texttt{xs} returns first \texttt{n} items
- What is the function \((\texttt{take} \ 3)\)?
- \texttt{include} \ x \ \texttt{ys} returns true if \texttt{x} is in the list. What is the function \((\texttt{include} \ 5)\)
So why do we want a curried function?

- Suppose we had already defined add’ and had the need to add 5 to every element of a list.
- Doing something to every element is a list is a common need. It is called “mapping”
- Instead of creating a separate function to add five, we can call
  
  ```haskell
  map (add’ 5) [1,2,3,4,5]
  ```
  or even
  
  ```haskell
  map (+5) [1,2,3,4,5]
  ```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

```haskell
> (1+) 2
3

> (+2) 1
3

> map (50 `div`) [10..16]
[5,4,4,3,3,3,3]

> map (`div` 25) [14,43,50,100]
[0,1,2,4]
```

In general, if $\oplus$ is an operator then functions of the form $(\oplus)$, $(x\oplus)$ and $(\oplus y)$ are called sections.
Curried Functions

- **Definition**: A function taking multiple parameters is *Curried* if it can be viewed as a (higher-order) function of a fewer parameters.
- Currying is good, since all functions can be viewed as having just a single parameter, and higher-order functions can be obtained automatically.

Haskell

- A fully-Curried lazy purely functional language with Hindley-Milner static typing. (*Fully-Curried* means all functions, including built-in arithmetic, are implicitly Curried.)
- Has many other features that make it perhaps the most advanced functional language available today.
Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

\[
\text{length} :: [a] \to \text{Int}
\]

for any type \(a\), length takes a list of values of type \(a\) and returns an integer.

Overloaded Functions

Type classes provide a structured way to control polymorphism.

\[
\text{sum} :: \text{Num } a \Rightarrow [a] \to a
\]

for any numeric type \(a\), sum takes a list of values of type \(a\) and returns a value of type \(a\).
Functor and typeclasses

- typeclass – a set of classes that can be used the same way: Examples Ord, Eq, Show. Typeclasses allow overloading (ad hoc polymorphism)

- Functor typeclass – a type that has fmap defined which can be applied to its members recursively (it can be mapped over)

- List is a member of the functor typeclass where

- **map**: (a->b) -> [a] -> [b]

- What if you wanted to “map over” something that wasn’t a list? You would need to create such a mapping function

Functor

data Tree a = Nil | Node a (Tree a) (Tree a)
deriving (Show,Eq)

instance Functor Tree where

fmap f Nil = Nil

fmap f (Node a x y) = Node (f a) (fmap f x) (fmap f y)

fmap (1+) Node 5 (Node 3 (Node 2 Nil Nil) (Node 55))
yields Node 6 (Node 4 (Node 3 Nil Nil) (Node 56)
Haskell properties

- Haskell is purely functional, so there are no variables or assignments

- Of course, there are still local definitions (in other words no value is being stored, we are just defining the pattern):
  
  ```haskell
  let x = 2; y = 3 in x + y
  ```

  or:
  ```haskell
  let x = 2
  y = 3 in x + y
  ```

- Note indentation in the previous code to get rid of the semicolon: Haskell uses a two-dimensional Layout Rule to remove extra syntax. Leading white space matters!

Analogy

Suppose you wanted to tell someone how to setup the hall for your 30th birthday party. Instead of giving the setup a name and recording it, it may be useful just to say “hey do this” without the formality of a name.
Lambda expressions (nameless function) can be used to avoid naming functions that are only referenced once.

For example:

\[
\text{odds } n = \text{map } f \ [0..n-1] \\
\text{where} \\
f \ x = x \times 2 + 1
\]

can be simplified to

\[
\text{odds } n = \text{map } (\lambda x \rightarrow x \times 2 + 1) \ [0..n-1]
\]

Haskell properties

- All expressions are delayed in Haskell:
  - \(\text{ones} = 1:\text{ones} \quad \text{can also write } [1,1..]\)
  - \(\text{ints}_\text{from } n = n : \text{ints}_\text{from } (n+1)\)
  - \(\text{ints} = \text{ints}_\text{from } 1 \quad \text{also: } [1..]\)
**Type inference** refers to the ability to deduce automatically the type of a value in a programming language. Parameter types aren’t required to be declared – so must infer them. We want to infer the most general type. Hindley-Milner (or “Damas-Milner”) is an algorithm for inferring value types based on use.

(a) come up with a list of constraints
(b) unify the constraints

Example:

len [] = 0
len (x:xs) = 1 + len xs
--len is expecting a list and returns a number

bar (x,y) = len x + y
-- bar must be expecting a (list, Num) as x is sent to len and y is added to the return value.

 What can you infer about type?

addPairs [] = []
addPairs ((x,y):xs) = (x+y): addPairs x
What can you infer about type?

addPairs [] = []
addPairs ((x,y):xs) = (x+y): addPairs x

- addpairs: [a] ->[b] (from first line)
- elements are the return set are numeric (as they are added)
- Elements of the first set are tuples
- numeric type of b is same as that of a
- (Num a) => [(a,a)] -> [a]

What can you infer about type?

del x [] = []
del x (y:ys)= if x==y then del x ys else y:(del x ys)
What can you infer about type?

\[
\text{del } x \; [] = []
\]
\[
\text{del } x \; (y:ys)\; = \; \text{if } x == y \; \text{then del } x \; ys \; \text{else } y:(\text{del } x \; ys)
\]

\[
\text{del } a \; [b] = [c] \; \text{(from first line)}
\]
a and b are the same type as they are compared for equality
c and b are the same type because y is an element of [c]
(Eq a) => a ->[a] - [a]

Strong typing

- Checks whether or not expressions we wish to evaluate or definitions we wish to use obey typing rules (before any evaluation takes place).

- A pattern is consistent with a type if it will match some elements of that type.
  - A variable is consistent with any type
  - A pattern (t:ts) is consistent with [p] if t is consistent with p and ts is consistent with [p]
Type Checking

\[ f :: \text{a->b->c} \]
\[ f \; x \; y \]
\[ \text{| g1 = e1} \]
\[ \text{| g2 = e2} \]
\[ \text{| otherwise = e3} \]

We must check
1. \( g1 \) and \( g2 \) are boolean
2. \( x \) is consistent with \( a \) and \( y \) is consistent with \( b \)
3. \( e1, e2, e3 \) are all of type \( c \).

Examples of type inference

- \( f \; (x,y) = (x, [\text{d'..y}]) \)
- What is the type of \( f \)?
- The argument of \( f \) is a pair. We consider separately the constraints of \( x \) and \( y \).
- \( y \) is used with ['d'..y] so it must be a Char.
- there are no restrictions on \( x \)
- \( f :: (a, \text{Char}) \rightarrow (a, [\text{Char}]) \)
Examples of type inference

- \( g(m,z) = m + \text{length } z \)
- What constraints are placed on \( m \) and \( z \)?
  M must be numeric, as used with +.
- \( z \) must be a list as used with length.
  Furthermore, since length returns an int, we assume \( m \) is also an int.

Unification

- We describe the intersection of the sets given by two type expressions.
- The unification of the two is the most general common instance of the two type expressions.

\[
\text{Unify} \\
(a, [a]) \text{ with } (\text{Int, } [b]) \\
\Rightarrow (\text{Int, [Int])}
\]
Unification need not result in a monotype.

- \((d,[d])\) and \(([b],c)\) unify as \(([b], [[b]])\)

Some can’t be unified

- \((d,[d])\) and \(([b],[\text{Int}])\)
- \(d\) must be an Int but an Int can’t be a [b]