Lecture 07: HO Programming and Type Classes
- Curried Functions
- Folding
- Type Classes

Reading: Hutton Ch. 3 & beginning of 7

You should also look at the Standard Prelude in Appendix B!
Recall that **function slices** are created from infix functions/operators by giving one of the operands, and leaving the other out. The missing operand is a parameter – this turns a function of two arguments into a function of one argument:

Main> \((3^2)\)  
9

Main> \((^2) 3\)  
9

Main> \((3^) 2\)  
9

Main> \((\lambda y \to 3^y) 2\)  
9

\[\frac{\lambda x \to \lambda y \to x^y}{(\\lambda x \to x^2)}\] 3 2  
9

\[\frac{\lambda y \to 3^y}{3^2} \Rightarrow 9\]

\[\frac{\lambda y \to 3^y}{\lambda x \to x^2} \Rightarrow 9\]
But notice that what we are doing here is partially applying a function to one of its arguments, and then stopping halfway through and calling it a new function:

\[
(\lambda x \rightarrow (\lambda y \rightarrow x^y)) \; 3 \; 2 \\
=> \beta \\
(\lambda y \rightarrow 3^y) \; 2 \\
=> \beta \\
3^2 \\
=> 9
\]
HO Programming: Curried Functions

We can do this any time we want, with any lambda expression with more than one argument:

```
Main> \( \lambda x \rightarrow (\lambda y \rightarrow x^y) \) 3
```

```
Main> f 2
9
```

By **referential transparency**, this is the same as:

```
Main> (\lambda x \rightarrow (\lambda y \rightarrow x^y)) 3 2
```

```
9
```

except that we “froze” the computation after applying the first argument.
HO Programming: Curried Functions

This explains why the following are all completely equivalent:

\[
\begin{align*}
f \ x \ y \ z &= (x, y, z) \\
f \ x \ y &= \lambda z \rightarrow (x, y, z) \\
f \ x &= \lambda y \rightarrow (\lambda z \rightarrow (x, y, z)) \\
f \ x &= \lambda y \ z \rightarrow (x, y, z) \\
f &= \lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow (x, y, z))) \\
f &= \lambda x \ y \ z \rightarrow (x, y, z)
\end{align*}
\]

which is proved by the type: all these will have the same type:

\[
\begin{align*}
f &: a \rightarrow b \rightarrow c \rightarrow (a, b, c) \\
f &= \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow (a, b, c)
\end{align*}
\]

Notice how the type arrows line up with the arrows in the lambda expression! Not a coincidence!
It also explains why all functions can be thought of as unary (one-parameter) functions.

\[
f(x, y, z) = (x, y, z)
\]

\(f\) takes three arguments and produces a triple \((3, 'a', True)\).
**HO Programming: Curried Functions**

$f = \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow (x, y, z)$

- $f$ takes three arguments
- $3 \rightarrow f \rightarrow f'$
- $f'$ takes one argument and produces a function $f'$ of two arguments:
  $$f' x = \lambda y \rightarrow \lambda z \rightarrow (x, y, z)$$
- $'a' \rightarrow f' \rightarrow f''$
- $f''$ takes one argument and produces a function $f''$ of one argument:
  $$f'' y = \lambda z \rightarrow (3, y, z)$$
- True \rightarrow f'' \rightarrow (3, 'a', True)$
- $f''$ takes one argument and produces a value:
  $$f'' z = (3, 'a', z)$$
This also explains why function application is left-associative and the arrow (in lambda expressions OR in type expressions) is right-associative:

\[
\begin{align*}
\text{f 3 'a' True} & \quad \text{f :: a -> b -> c -> (a,b,c)} \\
& \quad \text{f = \( \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow (x,y,z) \)}
\end{align*}
\]

\[
\begin{align*}
(f 3) 'a' True & \quad \text{f :: a -> (b -> c -> (a,b,c))} \\
& \quad \text{f = \( \lambda x \rightarrow (\lambda y \rightarrow \lambda z \rightarrow (x,y,z)) \)}
\end{align*}
\]

\[
\begin{align*}
((f 3) 'a') True & \quad \text{f :: a -> (b -> (c -> (a,b,c))))} \\
& \quad \text{f = \( \lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow (x,y,z)))) \)}
\end{align*}
\]
NOTE carefully that these functions DO have the same type:

\[ g :: a \rightarrow b \rightarrow c \]
\[ h :: a \rightarrow (b \rightarrow c) \]

But these functions do NOT have the same type:

\[ g' :: a \rightarrow b \rightarrow c \]
\[ h' :: (a \rightarrow b) \rightarrow c \]
Fold (also called reduce) is another function which uses a function as a parameter. There are two versions foldr (foldr) and foldl (fold left).

Fold right takes a list (constructed with the cons operator : ) and effectively replaces the cons with a function of two arguments, and the empty list with an “initial value” to get the recursion started:

\[
\text{foldr } f \text{ } v \ [e1, e2, e3]
\]

\[
\begin{align*}
e1 & : (e2 : e3 : []) \\
\end{align*}
\]
Here is a version of foldr similar to that given in the Prelude:

\[
\text{foldr} \, : \, (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]
\[
\text{foldr} \ f \ v \ [] \quad = \ v
\]
\[
\text{foldr} \ f \ v \ (x:xs) = f \ x \ (\text{foldr} \ f \ v \ xs)
\]

Thus, to sum the elements of the list, we could write:

\[
\text{foldr} \ (+) \ 0 \ [2,3,4] \quad \Rightarrow \ 9
\]

\[
2 : (3 : 4 : []) \quad : \quad 2 + (3 + 4 + 0)
\]
Higher-order Programming Paradigms

Here are some other applications of foldr – it is actually more powerful than you might think at first!

Calculating the length of a list:

```
foldr add1 0 [2,3,4]
```

```
add1 x y = y + 1
```

```
2 : ( 3 : 4 : [] ) 2 `add1` ( 3 `add1` 4 `add1` 0 )
```

```
g : 
  `:
    `:
      2
        `:
          3
            `:
              4
                []

add1 => 3

2
  add1 => 2
    3
      add1 => 1
        4
          0
```
Higher-order Programming Paradigms

Here are some other applications of foldr – it is actually more powerful than you might think at first!

Reversing a list:

\[ \text{snoc} :: a \rightarrow [a] \rightarrow [a] \quad \text{-- snoc is “cons” reversed} \]
\[ \text{snoc} \ x \ \text{xs} = \text{xs} ++ [x] \quad \text{-- because it adds to end instead of front} \]

\[
\text{foldr}\ \text{snoc} \ \text{[]} \ [2,3,4]
\]

\[
\begin{align*}
\text{foldr snoc [2,3,4]} & \Rightarrow \text{[4,3,2]} \\
\end{align*}
\]
Higher-order Programming Paradigms

Here is another applications of foldr – it is actually more powerful than you might think at first!

Collapsing a list:

\[ \text{foldr} \ (++) \ ([], \ [[2,3], \ [4,5], \ [6,7,8]]) \]

\[ [2,3] : [4,5] : [6,7,8] : [] \]

\[ [2,3] \text{++} [4,5] \text{++} [6,7,8] \text{++} [] \]

\[ [2,3] \text{++} [4,5] \text{++} [6,7,8] \text{++} [] \rightarrow [2,3,4,5,6,7,8] \]

\[ [2,3] \text{++} [4,5] \text{++} [6,7,8] \text{++} [] \rightarrow [2,3,4,5,6,7,8] \]

\[ [2,3] \text{++} [4,5] \text{++} [6,7,8] \text{++} [] \rightarrow [4,5,6,7,8] \]

\[ [2,3] \text{++} [4,5] \text{++} [6,7,8] \text{++} [] \rightarrow [6,7,8] \]

foldr (++) [] [ "hi ", "there ", "folks!" ] => "hi there folks!"
Type Classes and Overloading

An overloaded operator is the same symbol or name, but used for more than one type of argument:

\[ 2 + 4 \quad 3.4 + 5.6 \quad \text{also} \quad * \quad - \quad / \]

“hi” “ there” (Python)

True == False \quad 3 /= 5 \quad (Haskell)

Note that data or other syntax is sometimes overloaded

‘hi there!’ “hi there!” (Python)

34 can be Int Integer Float Double (Haskell)

Why do we do this? Flexibility and convenience and standard math practice!

Reading: Hutton Ch. 3.8, 3.9, 8.5

Note: there is really no difference between an “operator” and “function” – an operator IS a function, but usually is represented infix.
Type Classes and Overloading

Recall: A type is a set of related values and its associated operators/functions.

A type class is a set of types that share some overloaded operations/functions. In specific:

- The type class is defined by a set of data objects and the set of shared operators/functions;
- A type may be a member of multiple type classes;
- A type class may be a subset of another type class.

A type class is similar to an interface in Java: it defines what operations you can use with the type.

```java
// Queueable Interface
public interface Queueable {
    void enqueue(int n); // insert at the rear of the queue
    int dequeue(); // Remove and return head of queue
    int peek(); // Return head of queue without removing
    boolean isEmpty(); // returns number of integers in queue
    int size();
}
```
Type Classes and Overloading

**Example**: The type class `Eq` contains all the Equality Types, those that implement the equality operators:

```
==  /=
```

A type contained in a type class is called an instance of that class.

All types except for function types are instances of `Eq`.

**Reading**: Hutton Ch. 3.8, 3.9, 8.5
Naturally, these operators are polymorphic:

*Main> :t (==)
(==) :: Eq a => a -> a -> Bool

*Main> :t (/=)
(=/=) :: Eq a => a -> a -> Bool
Naturally, these operators are polymorphic:

*Main> :t (==)
(==) :: Eq a => a -> a -> Bool

*Main> :t (=/=)
(=/=) :: Eq a => a -> a -> Bool

*Main>

However, the polymorphism is restricted to types which are instances of Eq:

```
Eq a => a -> a -> Bool
```
class constraint

This says: “For any type `a` which is an instance of `Eq`, the function has type `a -> a -> Bool`”; any other type is forbidden.
Type Classes and Overloading

The type class **Ord** is a superset of **Eq**, and contains those types that can be totally ordered and compared using the standard relational operators:

\[
\begin{align*}
(\lt) & :\ Ord\ a \Rightarrow\ a\to\ a\to\ Bool \\
(\gt) & :\ Ord\ a \Rightarrow\ a\to\ a\to\ Bool \\
(\leq) & :\ Ord\ a \Rightarrow\ a\to\ a\to\ Bool \\
(\leq) & :\ Ord\ a \Rightarrow\ a\to\ a\to\ Bool \\
\text{min} & :\ Ord\ a \Rightarrow\ a\to\ a\to\ a \\
\text{max} & :\ Ord\ a \Rightarrow\ a\to\ a\to\ a
\end{align*}
\]
Type Classes and Overloading

The type class `Eq` is a superset of `Ord`, which contains those types that can be totally ordered and compared using the standard relational operators:

- `Ord` contains `Eq`
- `Eq` contains `==` and `/=`
- `Ord` contains `<`, `>`, `<=`, `>=`, `min`, `max`

Relational tests on tuples and lists is lexicographic:

```
*Main> "abc" < "abd"
True
*Main> "abc" < "abcd"
True
*Main> [2,3,4] <= [2,3,6]
True
*Main> [2,3,4] > [2,3]
True
*Main> [2,3,4] < [2,4,5]
True
*Main> ('a',5) < ('a',7)
True
*Main> (2,3) < (2,3,4)
```

The ordering on lists and tuples is also recursive:

```
*Main> [ [2,3], [2,4] ] < [ [2,3], [2,5] ]
True
```

<interactive>:202:9:  error:
  * Couldn't match expected type `(Integer, Integer)'
    with actual type `(Integer, Integer, Integer)'
  * In the second argument of `(<=)`, namely `(2, 3, 4)'
    In the expression: (2, 3) < (2, 3, 4)
  * In an equation for `it': it = (2, 3) < (2, 3, 4)
Type Classes and Overloading

The type class $\text{Eq}$ is a superset of $\text{Ord}$, which contains those types that can be totally ordered and compared using the standard relational operators.

Every instance of $\text{Ord}$ is an instance of $\text{Eq}$, i.e., $\text{Ord} \subseteq \text{Eq}$, which is similar to inheritance in Java and object-oriented languages:

```
Eq:  ==  /=
Ord:  < > <=|>=
      min max
```

Reading: Hutton Ch. 3.8, 3.9, 8.5
Type Classes and Overloading

**Num** – numeric types

The **Num** class contains numeric values, and consists of the following overloaded operators:

\[(+) :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a \rightarrow \ a\]

\[(*) :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a \rightarrow \ a\]

\[(-) :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a \rightarrow \ a\]

\[\text{negate} :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a\]

\[\text{abs} :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a\]

\[\text{signum} :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a\]

Hm... where is division?
Type Classes and Overloading

Integral – integer types

These are the instances of Num whose values are integers, and support integer division and modulus:

\[
\begin{align*}
\text{div} & : \text{Integral } a \Rightarrow a \rightarrow a \rightarrow a \\
\text{mod} & : \text{Integral } a \Rightarrow a \rightarrow a \rightarrow a
\end{align*}
\]

*Main> div 5 3
1
*Main> 5 `div` 3
1
*Main> mod 10 4
2
*Main> 10 `mod` 4
2
*Main>

Note that mod and div are prefix functions, to turn any function into infix, use back-quotes.
Type Classes and Overloading

**Fractional** – floating-point types

These are the instances of `Num` whose values are floating point, and support floating-point division and reciprocation:

```haskell
(/
) :: Fractional a => a -> a -> a

recip :: Fractional a => a -> a
```

```
*Main> 4.0 / 2.2
1.8181818181818181
*Main> recip 5
0.2
*Main> 4 / 2
2.0
*Main> 5 / 2
2.5
*Main> 5 / 2.2
2.2727272727272725
```

The symbols for integers are overloaded, so there is no "type-coercion" from integer to float here. The values are already fractional!