Topic 6: Partial Application, Function Composition and Type Classes

Recommended Exercises and Readings

- From Haskell: The craft of functional programming (3rd Ed.)
  - Exercises:
    - 11.11, 11.12
    - 12.30, 12.31, 12.32, 12.33, 12.34, 12.35
    - 13.1, 13.2, 13.3, 13.4, 13.7, 13.8, 13.9, 13.11
    - If you have time: 12.37, 12.38, 12.39, 12.40, 12.41, 12.42
  - Readings:
    - Chapter 11.3, and 11.4
    - Chapter 12.5
    - Chapter 13.1, 13.2, 13.3 and 13.4

Functional Forms

- The parameters to a function can be viewed in two different ways
  - As a single combined unit
    - All values are passed as one tuple
    - How we typically think about parameter passing in Java, C++, Python, Pascal, C#, …
  - As a sequence of values that are passed one at a time
    - As each value is passed, a new function is formed that requires one fewer parameters than its predecessor
    - How parameters are passed in Haskell
      - But it's not a detail that we need to concentrate on except when we want to make use of it

Curried and Uncurried Forms

- Uncurried form
  - Parameters are bundled into a tuple and passed as a group
  - Can be used in Haskell
    - Typically only when there is a specific need to do

- Curried form
  - Parameters are passed to a function sequentially
  - Standard form in Haskell

- Functions can be transformed from one form to the other

Curried and Uncurried Forms

- A function in curried form

```
multiply :: Int -> Int -> Int
multiply x y = x * y
```

- A function in uncurried form

```
multiplyUC :: (Int, Int) -> Int
multiplyUC (x, y) = x * y
```

Curried and Uncurried Forms

- Why use curried form?
  - Permits partial application
  - Standard way to define functions in Haskell
  - A function of n+1 arguments is an incremental extension of the form for a function with n arguments
- Why use uncurried form?
  - Syntactically similar to familiar languages
  - Helpful in some situations
    - Mapping / filtering a zipped list

Curried and Uncurried Forms

- A curried function can be transformed into an uncurried function
  - Call the uncurry function
    - Takes one parameter, which is a curried function
    - Returns one result, which is a function that performs the same task in uncurried form
- An uncurried function can be transformed into a curried function
  - Call the curry function
    - Takes one parameter, which is an uncurried function
    - Returns one result, which is a function that performs the same task in curried form

Uncurrying Examples

- Rewrite the following function in uncurried form

```
fma :: Int -> Int -> Int -> Int
fma a b c = a * b + c
```

- Compute the dot product of [1, 3, 5] and [2, 4, 6] in a single expression
  - Use map, (+), (*), uncurry, zip and foldr1

Partial Application

- Consider the function

```
plus :: Int -> Int -> Int
plus x y = x + y
```

- What's the type of plus?

Partial Application

- What happens when we pass plus parameters?
  - What's the type of plus 3 2?
- What if we only pass plus one parameter?
  - What's the type of plus 1?

Partial Application

- Partial application allows us to create new functions
  - Set one or more parameters to a specific value, leaving a function that still requires at least one parameter
  - Use the resulting function in any situation where a function is needed
    - Pass it to map, filter, foldl / foldr, zipWith, …
- Example: Write a function that adds 5 to every element in a list of Ints
  - Use map

Partial Application

- Generalize the function that adds a constant value to every element in a list
  - The amount to add will be passed as a second parameter

Operator Sections

- Writing a function to add two numbers together seems wasteful
  - The + operator already does this!
- An operator section is a partial application of an operator
  - Eliminates the need to write functions that duplicate existing operators
- What's the type of
  - (+)
  - (3 + 3)
  - (3 +)
  - (+ 3)

Operator Sections

- Use map and an operator section to add 5 to every element in a list

- Use map and an operator section to divide every element in a list by 5

- Use map and an operator section compute 5 divided by every element in a list

Operator Sections

- Forming a right section of (-) is problematic
  - (-3) is the integer minus 3, not a right section of the subtraction operator
- Solutions to this problem?
  - Use subtract in the Prelude module
    - Takes arguments in opposite order you would intuitively expect
    - subtract 4 2 is -2, not 2
    - subtract 3 is equivalent to right operator section of 3 and (-)
  - Or use (flip (-) 3)
- Example: Subtract 3 from every element in a list

Operator Sections Example

- Write a function that performs a Caesar Cipher
  - Apply a constant offset to every character in a string
  - The offset amount will be a parameter to the function
  - If adding the offset results in a character beyond 127, wrap around to the beginning of the ASCII table
  - Don’t use any helper functions

Partial Application in Other Languages

- Currying and partial application are now supported in Java
  - Added in Java 8 (March 2014)
  - Syntax isn't as clean as Haskell
  - Default is uncurried

```
import java.util.function.*;

class Curried
{
  public static void main(String args[])
  {
    Function<Integer,
          Function<Integer, Integer>> curriedAdd = a -> b -> a + b;

    // Form a new function with partial application
    Function<Integer, Integer> add5 = curriedAdd.apply(5);

    // Use curriedAdd to compute the sum of 5 and 3
    System.out.println(curriedAdd.apply(5).apply(3));

    // Use add5 to compute te sum of 5 and 3
    System.out.println(add5.apply(3));
  }
}
```

Partial Application in Other Languages

- Currying and partial application are supported in Python
  - Syntax isn't as clean as Haskell
  - Partial application can be applied to any function

Function Composition

- Consider our solution to the Caesar Cipher problem
  - Both the Haskell and Python solutions compute the correct result
  - But both have a potential concern, particularly for (very) long strings…

Function Composition

- Two (or more) functions can be combined (composed) into a single function
  - What if want to apply h to a parameter x, …
  - And then pass the result of that function application as a parameter to g, …
  - And then pass the result of that function application as a parameter to f?

  - Until now:

  - Another option:

Function Composition

- Example: Write a function that uses tail and reverse to remove the first and last elements from a list
  - Return the empty list if the length of the list is 2 or less

Function Composition

- Example: Consider a module for working with pictures
  - Assume that we already have a function, rotate90cw, that performs a 90-degree clockwise rotation of an image
  - How do we write a function, rotate90ccw, that performs a 90-degree counter-clockwise rotation of an image?

Function Composition

- The function passed to a higher-order function can be
  - A named helper function written by the programmer
  - A named function from a Haskell module
  - A lambda function
  - A partial application of function
  - An operator or operator section
  - A composition of any of the above

- Write several different expressions that add 2 to every element in a list of integers

Function Composition

- Improve the function that performs a Caesar Cipher
  - Access each element in the list only once
  - Do not create lists of intermediate results

Function Composition in Other Languages

- Java
  - Use the compose method

```
Function<Integer, Function<Integer, Integer>> cAdd = a -> b -> a + b;
Function<Integer, Function<Integer, Integer>> cFlipMod = a -> b -> b % a;

Function<Character, Character> encode = ((Function<Integer, Character>)(Compose::chr))
                                        .compose(cFlipMod.apply(128)).compose(cAdd.apply(1)).compose(Compose::ord);

Function<Character, Character> decode = ((Function<Integer, Character>)(Compose::chr))
                                        .compose(cFlipMod.apply(128)).compose(cAdd.apply(-1)).compose(Compose::ord);
```

Function Composition in Other Langauges

- There's no standard composition function in Python or C++
  - But it's possible to write your own…

Polymorphic Functions Revisited

- We have written polymorphic functions previously
  - Function's type signature includes type variables
  - Function can be applied to values of any type

- What if we want to write a function that works for some types but not others?

Polymorphic Functions Revisited

- Consider the following function

```
allEqual :: a -> a -> a -> Bool
allEqual x y z = (x == y) && (y == z)
```

- It doesn’t work correctly…
  - Haskell reports an compile time error…

Type Classes

- Type classes allow restrictions to be placed on type variables
  - Function can be applied to many types, but not all types
  - Restrictions are specified as part of the function's signature

- A correct version of allEqual:

Type Classes

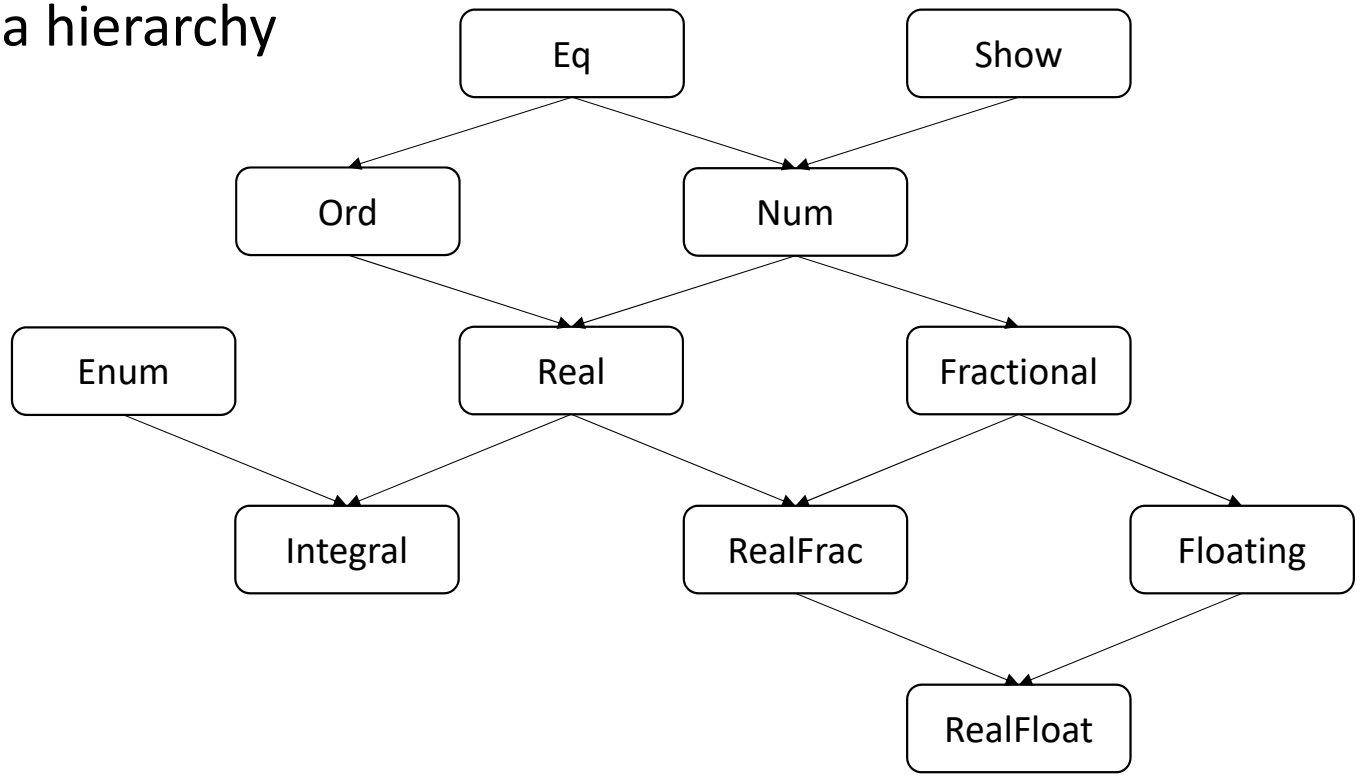
- Consider a polymorphic function that identifies the middle of 3 values
  - What operation(s) must be available?
  - How do we implement the function?

Type Classes

- Haskell provides numerous built-in type classes
  - Equality: Eq
    - Operations: ==, /=
  - Ordering: Ord
    - Operations: <, <=, >=, max, min, compare
  - Enumeration: Enum
    - Operations related to ..
  - Convert to String: Show
    - Operations: show :: a -> String, and others
  - Numeric: Num
    - Operations such as addition, subtraction, multiplication, negation, absolute value, …

Type Classes

- Type classes are a hierarchy



Type Classes

- We can create our own type classes
  - Allow us to impose our own restrictions on the types that can be passed to a polymorphic function
- Example: What if we wanted to restrict a function to types that can return their length (number of elements)?

Type Classes

- We can form a hierarchy among or own type classes
  - What if we want introduce another type classes that can be empty?
  - Such a type class could be a specialization of the HasLength type class

Type Classes

- Example: Create a polymorphic function that returns the status of a data structure that is an instance of CanBeEmpty
  - If the data structure is empty return "Empty"
  - If the data structure contains elements, return "Contains <n> elements"

Type Classes in Other Langauges

- In Java:
  - Classes are types
    - A method restricts the types to which it can be applied based on the static type of its parameters
    - This is much more restrictive than Haskell type classes because it requires a complete subtype relationship between the static type of the method and the dynamic type of the object being passed
    - This doesn't allow one to require multiple type classes
  - A method can require that it's parameter implements a particular interface
    - With generics, one can require that a parameter implement multiple interfaces

```
public <T extends Appendable & Comparable> void doIt(T x) {
   // Do something dependent on both appendable and comparable
}
```

Type Classes in Other Languages

- In Python
  - Type classes don’t exist
  - But some functions can still only execute successfully on some types
    - Parameters of any type can be passed to any function call
    - The function body uses whatever methods / operators are needed to perform its work
    - If they aren’t defined for the provided types then the program crashes

Summary

- Curried functions take their parameters individually
- Uncurried functions take their parameters as a tuple
- Partial application is possible with curried functions
  - A new function is created by fixing a subset of the parameters to an existing function
- Operator section: Partial application of an operator

Summary

- Function composition allows a new function to be created by combining (composing) several functions

- Type classes allow us to place constraints on the types used by polymorphic functions