An Introduction to Higher Order Functions

Braden Walters

Youngstown State University

11 February 2015

Outline

- Introduction
- A Brief Look at Referential Transparency and Side Effects
- First Look at Higher Order Functions
- Common Functions and Examples: Map / Fold / Filter
- Conclusion

Higher Order Functions

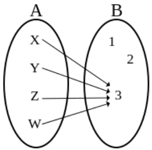
- In many programming languages, we think of the data we can pass around as being values or structures.
  - May be: integer, pointer, collection of integers and pointers, etc.
- We need to start thinking of functions as something that we can pass around.
- Higher Order Functions: A function which takes a function as a parameter, returns a function, or both.

Functional Programming

- HOFs and language features promoting them have been introduced into many popular programming languages and libraries.
  - Java 8, Python, Ruby, C#, and JavaScript, etc.
- Higher order functions are not the definition of functional programming!
- Functional programming is programming with functions as they are defined in mathematics.
  - Those $f(x) = y$ things.
  - Each function receives parameters and produces a result based only on those parameters.

Functional Programming

- We could say a function is a mapping from one set onto another. (Domain ->Codomain)



- Related concepts include:
  - Referential transparency and the elimination of side effects
  - Support for higher order functions (what I'll be talking about)

Referential Transparency and Side Effects

- Not necessary for our understanding of HOF, but nevertheless helpful.
- An expression has referential transparency if, provided we know the values of everything making up the expression, we can replace it with its computed result.

Referential Transparency and Side Effects

- Not necessary for our understanding of HOF, but nevertheless helpful.
- An expression has referential transparency if, provided we know the values of everything making up the expression, we can replace it with its computed result.
    - Example: Suppose $f(x) = 5 * x + 10$ and $g(y) = y - f(1)$.
        - We could remove $f(1)$ and rewrite $g(y) = y - 15$.
        - The expression calling $f(x)$ is referentially transparent.

Referential Transparency and Side Effects

- Not necessary for our understanding of HOF, but nevertheless helpful.
- An expression has referential transparency if, provided we know the values of everything making up the expression, we can replace it with its computed result.
  - Example: Suppose $f(x) = 5 * x + 10$ and $g(y) = y - f(1)$.
    - We could remove $f(1)$ and rewrite $g(y) = y - 15$.
    - The expression calling $f(x)$ is referentially transparent.
  - Counterexample: Consider C's *printf* function.
    - Suppose we replace a call to *printf* with the value it returns (*void*, i.e. nothing). The program will act differently.
    - *printf* produces a side effect: an expression calling this procedure is NOT referentially transparent.

Higher Order Functions in General

- Higher Order Functions - Functions that take functions as parameters and/or return a function.
- Frequently created so that we can write a gap in our code which the user of our function fills in.
    - Keeps our code generalised instead of specialised.
    - Example: Sort function lets user specify which items go first.
- Lambdas - Anonymous functions which can be written directly into expressions where they are used.
    - Often written directly into parameter lists of calls to higher order functions.

Lambdas in Popular Programming Languages

- Java 8: `higherOrderFunc(value, (AType a) -> a.attr);`
- Python: `higher_order_func(value, lambda a: a.attr)`
- Ruby: `higher_order_func value { |a| a.attr }`
- C# / Scala: `higherOrderFunc(value, a => a.attr);`
- Haskell: `higherOrderFunc value (\a -> attr a)`
- Clojure: `(higher-order-func value (fn [a] (get a :attr)))`

Map

- Suppose we need a function which takes a course and returns the full names of all students in that course.
    - Type signature of function:
      String [] getStudentNames(Course course)

Map

- Suppose we need a function which takes a course and returns the full names of all students in that course.
  - Type signature of function:
    String [] getStudentNames(Course course)
- We solve this imperatively by creating a string array and populating it with a loop.

Map

- Suppose we need a function which takes a course and returns the full names of all students in that course.
  - Type signature of function:
    String [] getStudentNames(Course course)
- We solve this imperatively by creating a string array and populating it with a loop.
- What's bad about this?
  - More verbose code is more difficult to comprehend at first glance.
  - Indexing a buffer allows bugs to arise.
  - Mutability leads to bugs.

Map

- It would be nice to have a function that just applies some function to all elements in a list.

Map

- It would be nice to have a function that just applies some function to all elements in a list.
- List<B> map(Function<A, B> function, List<A> input)

Map

- It would be nice to have a function that just applies some function to all elements in a list.
- List<B> map(Function<A, B> function, List<A> input)
- Assume our Course class has a member *students* of type *Student*[].

```
public String[] getStudentNames(Course course) {
  return Arrays.stream(course.students)
         .map((Student student) -> student.first_name + " "
                                   + student.last_name);
}
```

Map

- This approach is declarative. We describe what the solution is, not how you get there.
- Name in PLs: *map* in Haskell, Ruby, Clojure; *Select* in C#.

Map the Functors

- Things are about to become abstract! Deal with it. You're a computer scientist!
- The map function above doesn't just apply to lists and collections, but to all functors.
- Generalised map for functors (where F is anything which is a functor):

```
//Adapted from a presentation by Tony Morris.
//WARNING: This is not real Java code.
interface Functor<F> {
  <A, B> F<B> map(Function<A, B> function, F<A> functor)
}
```

Map the Functors

- Not so fast! Map MUST follow two laws for your type to be a functor:
  - If the transformation function returns its parameter unmodified, the functor is returned from map unmodified
  - Map should compose.

```
//Identity.
ftor.map((A a) -> a) == ftor

//Composition.
ftor.map((A a) -> f(g(a))) == ftor.map((A a) -> g(a))
                                  .map((B b) -> f(b))
```

Map the Functors

Function applied to cat:

Function mapped over cat in a cage functor:

Pictures by J. C. Phillipps

Map the Functors

- Why care about functors?
  - It turns out a lot of objects become really useful when you can map over them.

Map the Functors

- Why care about functors?
  - It turns out a lot of objects become really useful when you can map over them.
- The Option/Maybe type
  - The Option or Maybe type can be thought of as a type safe nullable type.
  - Initialised as either being some value or nothing.
  - How can we map over this?
    - If some value, apply the transformation to that value and return another option with the new value (some value ->some result)
    - If nothing, return the option unmodified (nothing ->nothing)

Option's Map

```
Option<String> opt1 = Option<String>.some("I'm an example");
Option<String> opt2 = Option<String>.nothing();

Option<String> res1 = opt1.map((String x) -> x + "!");
Option<String> res2 = opt2.map((String x) -> x + "!");
```

Option's Map

```
Option<String> opt1 = Option<String>.some("I'm an example");
Option<String> opt2 = Option<String>.nothing();

Option<String> res1 = opt1.map((String x) -> x + "!");
Option<String> res2 = opt2.map((String x) -> x + "!");
```

```
res1 == Some "I'm an example!"
res2 == Nothing
```

Option's Map

From Aditya Bhargava's Website: adit.io

Filter

- Suppose we want to find all students with failing scores.
    - Type signature of function:
      Student [] getFailingStudents (Course course)
    - The input course data structure should not be modified.

Filter

- Suppose we want to find all students with failing scores.
    - Type signature of function:
      Student[] getFailingStudents(Course course)
    - The input course data structure should not be modified.
- An imperative solution will either:
    - Copy the array and delete students with passing scores.
    - Create a new array and only copy students which have failing scores.
- In many languages, *foreach* can be used to eliminate the possibility of bounds errors.
- The code is still quite verbose though.

Filter

- List<A> filter(Function<A, bool> pred, List<A> input)
- Predicate (function returning boolean) is used to filter input.
- If the predicate returns true for an item, keep it, else remove it.

Filter

- List<A> filter(Function<A, bool> pred, List<A> input)
- Predicate (function returning boolean) is used to filter input.
- If the predicate returns true for an item, keep it, else remove it.

```
public Student[] getFailingStudents(Course course) {
  return Arrays.stream(course.students).filter(
    (Student s) -> s.score <= MAX_FAILING_SCORE);
}
```

- Name in PLs: *filter* in Haskell, Clojure; *select* in Ruby; *Where* in C#.

Fold

- Now suppose we want to know how many missing assignments there are in the class.
  - Type signature of function:
    int getTotalMissingAssignments(Course course)

Fold

- Now suppose we want to know how many missing assignments there are in the class.
  - Type signature of function:
    int getTotalMissingAssignments(Course course)
- Imperatively, we create an accumulator starting at 0 and iterate over all students, incrementing the accumulator.
- Not particularly bad, but mutating the data is necessary, which isn't really good.

Fold

- Now suppose we want to know how many missing assignments there are in the class.
  - Type signature of function:
    int getTotalMissingAssignments(Course course)
- Imperatively, we create an accumulator starting at 0 and iterate over all students, incrementing the accumulator.
- Not particularly bad, but mutating the data is necessary, which isn't really good.
- We need to accumulate a collection of values of one type into one value potentially of another type.

Fold

- The fold function reduces a list to one single value.
- A fold(Function<A, A, A> function, List<A> list)
    - If a list has only one item, return that item.

Fold

- The fold function reduces a list to one single value.
- A fold(Function<A, A, A> function, List<A> list)
    - If a list has only one item, return that item.
    - Fails on empty lists. Doesn't know what to return.

Fold

- The fold function reduces a list to one single value.
- A fold(Function<A, A, A> function, List<A> list)
  - If a list has only one item, return that item.
  - Fails on empty lists. Doesn't know what to return.
- A fold(Function<A, B, A> function, A seed, List<B> list)

Fold

- The fold function reduces a list to one single value.
- A fold(Function<A, A, A> function, List<A> list)
  - If a list has only one item, return that item.
  - Fails on empty lists. Doesn't know what to return.
- A fold(Function<A, B, A> function, A seed, List<B> list)
- Both functions take items from the list and aggregate them with the result of the previous step.
- The first step has no previous step.
  - Without a seed, start with the first two values in the list.
  - With a seed, the seed can be thought of as the result of the 0^{th} call.

Fold

```
public int GetTotalMissingAssignments(Course course) {
  return Arrays.stream(course.students).reduce(
      0 /*Initial value.*/,
      (Integer current_total, Student student)
      -> current_total + student.missing_assignments
    ).intValue();
}
```

- Name in PLs: *foldl*, *foldr* (with seed), *foldl*1, *foldr*1 (without seed) in Haskell; *reduce* in Ruby, Clojure; *Aggregate* in C#.

Solving Real Problems

- Begin using higher order functions immediately.
  - Start with the ones listed above: map, filter, fold.
  - Use some other higher order functions not mentioned, like sort.
  - Write your own.
- Entire procedures in your code can be made declarative (start your procedure body with the keyword *return* in Java/C#/Python).
- I use higher order functions frequently, and you should too!

Conclusion

Thank you. Now ask me questions!