Introduction

There are many things to study in order to gain an understanding and appreciation for functional programming. The idea behind functional programming is to program with a certain style that emphasizes the flow of data through a program as it becomes input and output to functions. Functional programming is a declarative programming paradigm in that it does not focus on individual steps taken to solve a problem, which is the focus of imperative programming. Declarative programming focuses on the computations involved in the solution and not necessarily when they are executed. Functional programming also differs from an object-oriented style that focuses on the different states and behaviors of objects involved in the problem. Instead, functional programming focuses on collections of data types and the functions defined on them.

There are many interesting topics related to functional programming. Aside from the main ideas involved, studying functional data structures can offer an understanding of how to reason about the flow of data in a functional platform. Some of the basic functional data structures that will be considered are lists, stacks, priority search queues and balanced binary search trees. Another data structure that will be considered is the zipper, which is more involved because it is a data structure that is derived from a defined recursive data structure. That is, given any recursively defined data structure, a zipper for that structure can be implemented. The zipper data structure incorporates the concept of state and location to a data structure so that the point of access to the data structure is the current state and not necessarily the original root element. The zipper data structure also has some intriguing mathematical underpinnings rooted in Category Theory, which will be introduced prior to the zipper. Category Theory is also present in many other areas of functional programming, one being the monadic programming pattern. Monadic programming is a design pattern that provides bindings between potentially complex computations. It offers functional solutions to binding functions that cannot be used as nested parameters through the concept of monads, which are a mathematical structure that contain specific mappings between specialized objects in the program.

Computability Theory is another area that is interesting because it offers a mathematical background to the idea of functions and whether they are computable. While not specific to functional programming, Computability Theory is a basis for many other studies, including the idea of lambda functions. Lambda functions are an object in Lambda Calculus, which formalizes the notion of function definitions and applications into the idea of lambda abstractions. Lambda functions are present in many programming languages and are a basis for functional programming. There also exists a theorem that binds the computability of a function with the existence of a specific lambda function, which itself binds the notion of computability with functional programming. All of these topics will be discussed in more detail. First, the basics of functional programming should be considered more carefully.

Basic Functional Programming

Functional programming is a commitment to a declarative style of programming whose goal is to concentrate on the data types of values and functions while minimizing side-effects. In comparison to common object-oriented programming, among many other programming paradigms, there are many notable differences. A large difference is that functional programming does not allow the use of mutable data. In other words, functional programming is stateless. At a higher level, it is helpful to compare the focuses of some different paradigms. The following are not necessarily mutually exclusive as it might be possible to combine the area of focus in an implementation.

1. An imperative style focuses on the individual steps that lead to a solution to the problem.
2. A declarative style focuses on the computations involved in solving the problem and not necessarily when they are computed.
3. An object-oriented style focuses on the state and behavior of the abstract notion of objects involved in leading to a solution.
4. A functional style focuses on the data flow through a program along with abstractions over them.

Even though the language of Haskell will be utilized here, this is not a tutorial for learning Haskell but rather a survey of some of the aspects of Haskell while discussing functional programming.

There are many differences between functional programming and object-oriented programming. One of the first, somewhat difficult, concepts is that a function is treated differently than in an object-oriented platform. That is, functions in Haskell are treated as instances of data type classes just as variables are instances of classes in Java. An immediate consequence of this is that functions can be defined to have as input and output function types so that they can utilize function inputs and outputs. This type of function is called a higher-order function, and the type of a function is not much different than that of a named value. An example of an integer in Haskell that is stored as a named value might look like:

```
x :: Integer,
```

and a function that takes as input a function over the integers and a function over the booleans might look like:

```
f :: (Integer → Integer) → (Bool → Bool).
```

This treatment of functions allows many interesting ways to operate on data structures. A classic data structure utilized in functional programming is a list, and a fun example of a higher-order function over a list is `map`, which takes a function and a list and returns the list with the function applied to every element of the list. As an example:

```
map (2*) [1,2,3,4].
```

This applies the function `(2*)` to every element in the list, returning `[2,4,6,8]`. The definition of this function might seem complicated, but in fact it is not difficult to implement:

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

The function `(2*)` might also seem like a strange function, but it is just a partially applied function. The function that multiplies two integers has the type:

```
(*) :: Integer → Integer → Integer,
```

and when the first integer, in this case 2, is supplied, it just returns another function with type:

```
(2*) :: Integer → Integer.
```

In fact, all functions take each parameter, return a type with one less parameter, and evaluate the next parameter recursively until each parameter is used.

There are many other functions that can be defined over a data structure such as a list, many of which can be implemented with only a few lines of code. Some other trivial examples are `filter`, which takes a predicate and a list and returns a subset of the list satisfying the predicate, and `fold`, which takes a function and a list and returns a single value that is the result of applying each element of the list to each other according to the function. Some of the difference between functional and object oriented programming can become evident when comparing implementations of functions and Haskell and Java. Here is a Haskell implementation that assigns the function to that of a partially applied function:

```
filterNeg = filter (>=0),
```

and here is a Java implementation:

```
public ArrayList<Integer> filterNeg(List<Integer> nums) {
  List<Integer> filtered = new ArrayList<Integer>();
  for (Integer n : nums)
    if (n >= 0) filtered.add(n);
  return filtered;
}.
```

From this, it is easy to see that implementing functions in a functional style might often be more pleasant than in other styles. The Haskell code is easy to read and understand while the Java code may cause a prolonged understanding of the function performed, especially in the case of more complex functions with non-descriptive names.

One of the basic concepts in a functional setting that is important is that of type classes. Type classes drive the overall implementation of the program. A type class is a collection of types that that are required to respond to certain functions that are in the definition of the type class. A type class in Haskell can be thought of as analogous to an interface in an object-oriented language; however, this might be a dangerous analogy because type classes focus on data and interfaces focus on an object's state and behavior. Each are still structures that drive the entirety of the design of the program in each type of programming style.

A type class in Haskell is a list of functions that may or may not be given implementations. A data type is said to derive from a type class if it is desired to be substitutable for any generic type that is an instance of that class. A data type that derives a type class is required to provide implementations of the functions specified in the type class definition so that the functions can behave accordingly with respect to the specific data type. The simplest example of a type class is: `Eq a`, which specifies the functions that check for equality and non-equality:

```
class Eq a where
  (==), (/=) :: a → a → Bool
  x==y = not (x/=y)
  x/=y = not (x==y).
```

In this example, default implementations have been given that require a data type deriving this type class to implement only one of the functions specified. This class will automatically define the other function as the negative of the one defined by an instance. A couple examples of data types that are defined to derive this class are integers and booleans so that they can be tested for equality:

```
instance Eq Integer where
  n==n = True

instance Eq Bool where
  True==True = True
  False==False = True.
```

In these examples, if two integers are tested for equality and they are not the same integer, the default definition of `(/=)` will cause a value of `False` to result. This will also be the case if two booleans are tested that are not the same. A couple more trivial examples are:

```
class Visible a where
  toString :: a → String
  size :: a → Integer

class (Eq a) => Ord a where
  (<), (<=), (>), (>=) :: a → a → Bool
  max, min :: a → a → a
  compare :: a → a → Ordering
data Ordering = EQ | LT | GT.
```

An immediate advantage is that functions can be defined over generic types even if they utilize functions specific to a certain type class:

```
max :: Ord a => a → a → a
max x y = if x>y then x else y.
```

This is another step in abstracting from specific types and the focus to be moved from specific data types to the flow of generic data as long as the data reacts to certain functions. In other words, this is Haskell's mechanism for promoting the use of polymorphic data types.

At this point, it might be natural to inquire whether it is possible to define type classes that take other type classes, as instances, instead of data types, creating yet another layer of abstraction and promoting the use of polymorphic type classes. This is useful tool and is known as a type constructor. A common and useful example is the `Monad m` type constructor. This is a useful mechanism for providing functional solutions to sets of problems that do not lend themselves naturally to the functional setting. An example of an operation that is not inherently functional or declarative is input and output. This will be discussed in the section on monadic programming.

Elementary Data Structures

An important exercise when studying new programming paradigms is to implement some data structures. The implementations of data structures are often elegant in Haskell. One of the most basic data structure is a list and is useful in the implementations of other trivial data structures.

```
data [a] = [] | a : [a]

null [] = True
null _ = False

length [] = 0
length (x:xs) = 1 + (length xs)

head [] = Error
head (x:_) = x

tail [] = Error
tail (x:xs) = xs

init [] = Error
init (x:xs)
  | null xs = []
  | length xs == 1 = [x]
  | otherwise = x : (init xs)

last [] = Error
last (x:[]) = x
last (x:xs) = last xs.
```

Many more useful functions can be defined over lists; however, these are the main functions that are useful in implementing other data structures. The existence of the list allows basic data structures to be implemented in a simple manner. All of the following data types are recursively defined but using lists to define them will reduce code duplication because all of the basic functions are only implemented in the list data structure and not re-implemented in every new data structure.

Other classic data structures include stacks, queues, priority search queues and balanced binary search trees. Here is an implementation of a stack:

```
data Stack a = S [a]

push (S xs) x = S (x:xs)
```

```
pop (S []) = Error
pop (S xs) = (head xs, S $ tail xs)

top (S []) = Error
top (S xs) = head xs

isEmpty (S xs) = null xs

emptyStack = S []

size (S xs) = length xs

toStack xs = S xs.
```

A queue is pretty similar to that of a stack:

```
data Queue a = Q [a]

enqueue (Q xs) x = Q (xs ++ [x])

dequeue (Q []) = Error
dequeue (Q xs) = (head xs, Q $ tail xs)

front (Q []) = Error
front (Q xs) = head xs

back (Q []) = Error
back (Q xs) = last xs

isEmpty (Q xs) = null xs

emptyQueue = Q []

size (Q xs) = length xs

toQueue xs = Q xs.
```

A deque is another data structure that combines the ideas of the stack and queue in that it adds functionality for adding and removing elements from both the front and the back of the deque. It is nothing more than a combination of the stack and queue implementations. A priority search queue is another variant of a queue data structure but with the existence of priorities assigned to each element. The priorities affect which position in the queue that the element is inserted. Elements with a higher priority are inserted near the front of the queue, and elements with a lower priority are inserted near the rear of the queue. Here is a basic example of a priority search queue, where the elements and their priorities are stored as pairs in the data structure:

```
data Pqueue a p = P [(a,p)]

insert (P xs) x prior = P ( [(a,ap) | (a,ap)<-xs, ap>prior]
                         ++ [(x,prior)]
                         ++ [(b,bp) | (b,bp)<-xs, bp<=prior] )

insertHighest (P []) _ = Error
insertHighest (P xs) x = insert (P xs) x (snd $ head xs)

insertLowest (P []) _ = Error
insertLowest (P xs) x = insert (P xs) x (snd $ last xs)

removeHighest (P []) = Error
removeHighest (P xs) = (fst $ head xs, P $ tail xs)

removeLowest (P []) = Error
removeLowest (P xs) = (fst $ last xs, P $ init xs)

highestElem (P []) = Error
highestElem (P xs) = fst $ head xs

lowestElem (P []) = Error
lowestElem (P xs) = fst $ last xs

highestPrior (P []) = Error
highestPrior (P xs) = snd $ head xs

lowestPrior (P []) = Error
lowestPrior (P xs) = snd $ last xs

isEmpty (P xs) = null xs

emptyPqueue = P []

size (P xs) = length xs

toPqueue xs prior = P [(x,prior) | x<-xs].
```

The balanced binary tree is a hurdle in the topic of basic data structures. The implementation of it in Haskell highlights some more advantages in Haskell in the context of pattern matching. All of the previous examples have utilized pattern matching because each definition is given multiple implementations. The first implementation that matches the parameters supplied at runtime is the one that is executed. For example, almost every function checks for an empty list in the first implementation. The empty brackets coincide with the definition of an empty list so if a parameter is given that contains an empty list, the first definition will match the pattern and execute a general error that is supposed to have been defined accordingly. The `init` function in the first definition of the list data type also uses pattern matching in the form of three guards. In that example, the first guard that matches the parameter given will be the one that is executed. The implementation of the balanced binary tree will include some helper functions that rely heavily on guards and pattern matching to perform the rotations to balance the tree. Here is the implementation of a balanced binary tree:

```
data BBTree a = Nil | Node a (BBTree a) (BBTree a)

insert Nil x = Node x Nil Nil
insert (Node x t1 t2) y
  | y<=x = let t = Node x (insert t1 y) t2 in balance t
  | otherwise = let t = Node x t1 (insert t2 y) in balance t

remove t y = toBBTree (filter (/=y) $ toList t)

exists Nil _ = False
exists (Node x t1 t2) y = y==x ||
  if y<=x then exists t1 y else exists t2 y

size Nil = 0
size (Node _ t1 t2) = 1 + size t1 + size t2

depth Nil = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

empty Nil = True
empty _ = False

emptyTree = Nil

toBBTree [] = Nil
toBBTree (x:xs) = insert (toBBTree xs) x

toList Nil = []
toList (Node x t1 t2) = toList t1 ++ [x] ++ toList t2

-- Private helper methods.

balance Nil = Nil
balance (Node x t1 t2)
  | depth t1 > depth t2 = leftHeavyBal (Node x t1 t2)
  | depth t1 < depth t2 = rightHeavyBal (Node x t1 t2)
  | otherwise = Node x (balance t1) (balance t2)

leftHeavyBal (Node x (Node y s1 Nil) t2) =
  Node y (balance s1) (balance $ Node x Nil t2)
leftHeavyBal (Node x (Node y s1 (Node z u1 u2)) t2)
  | depth s1 > depth (Node z u1 u2) = -- Single right rotation.
    Node y (balance s1) (balance $ Node x (Node z u1 u2) t2)
  | otherwise = -- Left-Right rotation.
    Node z (balance $ Node y s1 u1) (balance $ Node x u2 t2)

rightHeavyBal (Node x t1 (Node y Nil s2)) =
  Node y (balance $ Node x t1 Nil) (balance s2)
rightHeavyBal (Node x t1 (Node y (Node z u1 u2) s2))
  | depth (Node z u1 u2) < depth s2 = -- Single left rotation.
    Node y (balance $ Node x t1 (Node z u1 u2)) (balance s2)
  | otherwise = -- Right-Left rotation.
    Node z (balance $ Node x t1 u1) (balance $ Node y u2 s2).
```

The balancing in the binary tree is trivial in that the specific cases of the tree are pattern matched, which exposes the values of each node and allows them to be used in reconstructing and returning the newly balanced tree. For example, in the balancing functions, the different data points in the nodes are decomposed into variable names that are then rearranged in the construction of the balanced nodes.

Another interesting data structure is the zipper data structure. The zipper structure is a reconstruction of any recursively defined data structure to a new one that allows for the concept of a present location. Zippers will be discussed in detail after learning some Category Theory as there are some connections between the type of a zipper and the process of differentiation in mathematics. This is understood by considering data types and programs as collections of objects and morphisms in a category.

A Little Category Theory

Category Theory is the study of general mathematical structures that consist of objects and functions between those objects. This section is going to take a look at some of the very basic concepts in Category Theory in order to build a basis for the concepts that will be used when discussing zippers and monads. An understanding of Category Theory is not necessary in order to understand zippers and monads; however, many of the concepts in Haskell have counterparts rooted in less trivial concepts in Category Theory, some of which will be introduced in the following sections.

As with any discipline of mathematics, Category Theory is a list of definitions and theorems based on those definitions. Category Theory generalizes the study of mathematical structures by studying structures with only two requirements: the properties of identity and associativity. The types of structures span many areas including Number Theory, Calculus and Analysis, Linear Algebra and Differential Equations as well as Topology.

A category `C` consists of three things: a set `Ob(C)` of objects, a set `Mor(C)` of morphisms that act on those objects and a function that represents composition between every element of `Mor(C)`. The notion of composition will be represented as concatenation. For each morphism `f:X → Y` and `g:Y → Z` in `Mor(C)`, where `X,Y,Z` is in `Ob(C)`, the morphism `fg: X → Z` is in `Mor(C)`. The following two properties must also hold:

1. For all `f,g,h` in `Mor(C), f(gh) = (fg)h.`
2. For all `X` in `Ob(C),` there exists an identity morphism `1_X` such that `1_X(X) = X.`

An example of a category is the category `Set,` which includes all sets and all functions between those sets. Another example of a category is `Grp,` which is the category of all groups and all group homomorphisms. Other examples include:

1. All partially-ordered sets with monotonic functions.
2. All directed graphs where vertices are objects and edges are morphisms.

3. All topological spaces with continuous maps.
4. All metric spaces with all metric functions.
5. Vector spaces with linear maps & modules with module-homomorphisms.
6. Smooth manifolds with differentiable maps.
7. All categories with functors.

Another central idea to many areas of mathematics is preserving structure. For example, a homomorphism in algebra preserves the structure between groups and a homeomorphism in topology preserves the structure between two topological spaces. The notion of a structure-preserving map between categories also exists.

Let C, D be two categories. A functor F is a mapping $F\colon C \to D$ with the property that for every unique object X in $Ob(C)$, FX is an associated unique object in $Ob(D)$. Also, for every morphism f in $Mor(C)$, there is an associated morphism Ff in $Mor(D)$. Here, $Ff\colon FX \to FY$ if $f\colon X \to Y$ where X, Y are in $Ob(C)$. The following two properties must also hold, for all objects X and morphisms f, g:

1. $F1\_X = 1\_FX$
2. $F(fg) = (Ff)(Fg)$.

Some examples of functors include:

1. The power functor $P\colon Set \to Set$, which maps every element of a set to its power set.
2. The forgetful functor is a functor that removes elements from a category. Examples include the functor which sends the category of groups to the category of sets. Another example is the functor that sends the category of vector spaces to the category of modules.
3. Any group can be considered a category with a single object being the group and the morphisms being the elements of the group. Then the group action is a functor. A trivial group action is the group homomorphism. Another example is that of the special linear group

`SL(2,R)` on the complex plane resulting (elliptic, parabolic and hyperbolic) isometries.

4. An identity that maps every object and morphism to itself is a functor.

Another important concept is a special case of a functor is an endofunctor, which maps a category to itself.

A natural transformation is a mapping between two functors. Let `F,G: C → D` be two functors. A natural transformation `n` is a mapping `n: F → G` such that for every object `X` in `Ob(C)`, the natural transformation is defined as `n_X: FX → GX` in `Mor(D)`. The property `(n_Y)(Ff) = (Gf)(n_X)` must also hold for every morphism `f: X → Y` in `Mor(C)`. Just as a functor is a structure-preserving map between two categories, a natural transformation associates each element in a functor-evaluation of a category with another functor-evaluation of the same category. Therefore, a natural transformation is a structure-preservation of functor-evaluations.

The final definition is the category theoretic definition of a monad, which is equivalent to monads in Haskell. In fact, it can be shown that the notion of monad in Haskell is derived from functor type classes. Haskell monads also satisfy the properties that are required by the category theoretic definition. If `C` is a category, then a monad consists of an endofunctor `T: C → C` and two natural transformations `n: id_C → T` and `u: TT → T`, each of which satisfy the following properties:

1. `u(Tu) = uu`
2. `un = u(Tn) = id_T`

These two properties are equivalent to the common two properties that are used to preserve structure in both the definition of a category and the definition of a functor. The first property means that it does not matter whether the functor is evaluated and then the natural transformation, or vice-versa. This is equivalent to associativity. The second property is equivalent to the identity element between two natural transformations. In the section on monadic programming, it will be shown that the type constructor equivalent to the monad will derive from functors and will satisfy these two properties.

The Zipper Data Structure

The zipper data structure is a special-purpose structure that an be constructed from any recursively defined data structure. The goal of the zipper of a recursively defined structure is to implement a present position to a data structure. Thinking in terms of a general tree, which might have a type that looks like:

```
data Tree a = Leaf a | Node a [Tree a],
```

a present position can be implemented if this tree is used in the context of a maze for a game. The start location for the maze would be the root node, rooms without any doors leading to new rooms (dead-ends) are leafs and other rooms are nodes containing doors to other rooms which are recursively defined trees. The generic data type `a` can be used to represent items in the maze so each room contains a list of items. The root node of this structure is the only point of access. Therefore, if the player travels deep into the maze and decides to move items around, the programmer, only having access to the maze through the point of access, will have to traverse the tree every time a change is made to a node in the tree. Computationally, this is more heavy-handed than needed.

The idea behind the zipper is to alter the point of access to the player's current position in the maze. This is accomplished by changing the data structure to a pair. The first coordinate will contain the node with the current position as well as a recursive backwards path from the current position to the original root node. The second coordinate contains recursive aspect of the current position, which is in this case the subtree. In the context of the maze, the first coordinate contains the current room along with the backwards path which leads back to the entrance. And the second coordinate contains all doors available to the player in the current room. There is also no information lost in the translation of the original tree to the zipper of the tree because all paths not taken are just encapsulated in the backwards path in the first coordinate of the zipper. The performance characteristics of altering the items in the current room are better when using a zipper because there is no requirement to traverse the tree to the current player's location to change the list of items. It is also easy to move between rooms as the only operation that takes place when moving between rooms is adding/removing single nodes between the two coordinates in the zipper structure.

A simpler data structure that yields a simple zipper structure is a list. One of the ways that a list can be defined is:

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

which is either empty or contains an element and another list. The entrance to a list is the head element. The current location would contain a value. The backwards path from the current location to the head element would just be a list that goes from the current element back to the head. This information is encapsulated in the first coordinate of the zipper, and the second coordinate would contain the sublist of the current location. As an example the list `[1,2,3,4,5,6]` with the current element being `3` in the zipper would be `([3,2,1], [4,5,6])`. So the data type of the list's zipper would be:

```
data ZipperList a = (List a, List a).
```

Moving the current position in the list's zipper reduces to popping and pushing elements between the two lists in each coordinate. Another example is a binary tree, which can be defined to have the type:

```
data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a).
```

Traversing a binary tree means that a binary decision is made at each node. Therefore, the current location is a `BinaryTree` and the backwards path is a list of the nodes traversed as well as the nodes not traversed. So each element in the backwards path will require a list of decisions of `BinaryTrees`, and the second coordinate in the zipper will just be a `BinaryTree`:

```
data Branch a = Left a (BinaryTree a) | Right a (BinaryTree a)
data ZipperBinaryTree a = ([Branch a], BinaryTree a).
```

Functions to move through the zipper are as follows:

```
goLeft :: ZipperBinaryTree a → ZipperBinaryTree a
goLeft (_, Leaf val) = Error
goLeft (path, Node val left right) = (Left val right : path, left)

goBack :: ZipperBinaryTree a → ZipperBinaryTree a
goBack (Right val : [], _) = Error
goBack (Left val right : path, left) = (path, Node val left right)
goBack (Right val left : path, right) = (path, Node val left right).
```

The function `goRight` is easy to implement compared to `goLeft`. A more interesting zipper is one for a general tree:

```
data Tree a = Leaf a | Node a [Tree a]

data Branch a = Selection a [Tree a] [Tree a]
data ZipperTree a = ([Branch a], Tree a)

goDown :: ZipperTree a → Integer → ZipperTree a
goDown (_, Leaf val) _ = Error
goDown (path, Node val children) index =
  (Selection val (take index children) (drop (index+1) children),
   children!!index)

goBack :: ZipperTree a → ZipperTree a
goBack (Selction val [] [] : [], _) = Error
goBack (Selection val left right : path, subtree) =
  (path, Node val (left ++ subtree ++ right))

modify :: ZipperTree a → a → ZipperTree a
modify (Selection val left right : path, subtree) newVal =
  (Selection newVal left right : path, subtree).
```

In terms of Category Theory, there is a connection between a recursively defined data structure's type and it's zipper's data type. In fact, the type of the zipper can be derived via differentiation of a polynomial representation of the recursive type. If the set of all data types are viewed as objects and the set of all programs utilizing those types are morphisms, then these two sets can form a category if a data type's identity is the program that sends each element of the data type to itself and if the programs obey associativity. A data type `X` that is expressible as a polynomial functor is defined in polynomial form as `FX` where `F` is a functor. The symbol `d` will be used to denote differentiation. The derivative `(dF)X` represents the one-hole contexts of the data type `X`. The one-hole context is the data type with single substructure removed, which makes sense because differentiation introduces a hole in the structure being differentiated. In this case, the substructure that is removed is a single hole because the substructure is accessible via a single access point in the recursive definition. The symbol `(dF)X` can also be written `d_X(FX)` which fixes the point at which to differentiate. This allows the derivative operator to be applied to the data structure `FX` at the point `X` instead of the polynomial functor `F`.

The one-hole context will help yield the data type for the zipper. Note that in recursive data types, the data type can be equated with the recursive aspect of its definition. This means that for a data type `X`, `X ~ FX`. In other words, the data type and it's polynomial functor definition are isomorphic. Therefore, it is also true that `dX ~ (dF)X ~ d_X(FX)`. From this, the derivative of the polynomial functor yields a one-hole context for the data type `X`. The neglected substructures are collected together by the one-hole context; therefore, each one-hole context calculation collects a single substructure and removes a substructure from the tree, and the one-hole context of the zipper is then a list of the one-hole contexts formed by the derivative. In words, the zipper type for the data type `X` is given by `(List dX, X)`, according to the rules of differentiation of polynomial functors and their equivalence to the data type, above.

As a first example, recall the data type for a list:

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

The polynomial functor is representable as `F_A(X) = 1 + AX`, where the addition represents the disjunction in the data type and the multiplication represents the conjunction of the data value and the recursive declaration in the cons cell. From the previous paragraph, `List(A) ~ F_A(X) ~ F_A(List A)`, therefore `dX ~ (dF_A)X ~ d_X(1 + AX) = A`. Since the one-hole context is a single A, the zipper for the list type is `(List A, List A)`. Recall the binary tree:

```
data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a).
```

The polynomial functor is `F_A(X) = A + AX^2` and `(dF_A)X = 2AX` so the zipper for the binary tree has type `(List 2AX , BinaryTree A)`, which is equivalent to the previous implementation of the binary tree's zipper. Recall that the first coordinate the binary tree's zipper was a list of binary decisions, each of which contained a value and a recursive declaration. This is equivalent to `List(2AX)` because this contains an addition of two conjunctions of data values and data types. Recall that addition corresponded to disjunction. This will also appear in the case of the general tree, which has type:

```
data Tree a = Leaf a | Node a [Tree a].
```

The polynomial functor is `F_A(X) = A + AX^n`, where `n` represents the number of elements in the list. The derivative is `(dF_A)X = nAX^(n-1)` and the zipper type is `(List nAX^(n-1), Tree A).`

The zipper data structure is a nice example of a functional data structure because it is trivial to understand how to implement, and it ties together important concepts in Category Theory, which is a basis for many of the concepts and type classes that exist in Haskell. Another concept that is not exceptionally nontrivial but involves category theoretic concepts and enables many useful patterns in programming are monads.

Monadic Programming

Monadic programming refers to programming with the use of monads. Monads are a simply a design pattern that provide a function solution to binding between the output of one function and the input of another function. The definition of a monad in Haskell is equivalent to a monad in Category Theory in that they both satisfy the same properties of natural transformations.

The type `Monad m` in Haskell is a polymorphic type constructor. This means that it is in the form of a type class, but `m` is itself a type class so instances of the type constructor provide type classes and build another type class. Here is the basic Haskell interpretation:

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b.
```

Instances of this type constructor transform elements of the type `m a` to the type `Monad` m. The function `return` moves an element of the initial type `a` to an element of the type `m a`, and the function `(>>=)`, called the bind function, takes an element of type `m a` as well as a function that moves the element of type `a` to the type `m b`, resulting in an element of type `m b`. For example:

```
g >>= f
```

performs the function `g` and uses the output of `g` as input to `f`. From this explanation, it seems that this could be written as `f g`, so the input to `f` is the function `g`. This will not work and is an important point to understand. The ladder form means that the input type to `f` must match the type of the function g and not just its output type. Therefore, the output of `g` needs to be binded to `f` through the use of another function. The generalization offered in the type constructor `Monad m` does not specify anything about the functions that are bound together. These two functions can be very complex entities, which makes this a powerful technique.

Different instances of `Monad m` may require different implementations of the functions, but the general idea is that the binding operator behaves with the following general definition:

```
g >>= f = do x ← g
             f x.
```

A simple example of use of this operation is to extract input from the standard input device and print it to the standard output device. In Haskell, two functions that perform input and output are `getLine` and `putStrLn`. A function that binds the input to the print function would simply be:

```
getLine >>= putStrLn.
```

The types of these functions are `getLine :: IO String` and `putStrLn :: String → IO ()`. The function `getLine` is of type that returns a string from the standard input, and the function `putStrLn` takes in a string and returns an output action but does not return a specific value, which is encapsulated in the `()` symbol. If, instead of using the bind function, the program were written as `putStrLn getLine`, an implication is being made that the type of `putStrLn` requires `IO String` and not just `String` as its input. The bind operator is allowing any type class `m String`, provided the type constructor `Monad m` is implemented for that particular type class, to be bound to the input of `putStrLn`, enabling much more freedom.

There are many trivial examples of monads. One of them is the ID monad, which offers a binding between an identity data type and a function.

```
data Id a = Id a

instance Monad Id where
  return = Id
  Id x >>= f = f x.
```

In this example, the `return` function returns an element of the type class `Id a`, and the bind function extracts the element from the data type `Id a` and uses that element as input the function provided.

The data type `Maybe a` is used to provide error checking for values of type `a`. The data type is defined as

```
data Maybe a = Nothing | Just a.
```

Functions can be defined to output this type so that calculations that would result in an error would just assign the value of `Nothing` as output. Calculations with output would just be wrapped in the `Just` construct. Here is an example of a function that uses this type:

```
safeDivide :: Integer → Integer → Maybe Integer
safeDivide _ 0 = Nothing
safeDivide a b = Just (a/b).
```

A monad for the type class can be defined as follows:

```
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  Just x >>= f = f x.
```

This is similar to `Monad Id` but provides an overloaded implementation of the bind function, depending on the type of value provided. With this, the definition of `safeDivide` can be redefined. If the value `y` has type `Maybe Integer` and `safeDivide x y = x/y`, then the function will become:

```
y >>= (safeDivide x).
```

Another example of a monad is the list monad. The original goal of the monad is to create types from types, with the application of some function that is used in the bind function. The list monad is used to create lists from lists:

```
Instance Monad [] where
  return x = [x]
  xs >>= f = concat . map f $ xs.
```

In this list monad, the `return` function acts as expected; however, the bind function is interesting. The function `f` in this case is applied to every element of the list `xs`, and, for each element, it creates a list of elements as its result. Therefore `(map f) xs` creates a list of lists, and the function `concat` concatenates them all into a single list. For example, here is an implementation that returns a list of all the squares and cubes from a list of integers:

```
calcSquaresCubes :: Integer → [Integer]
calcSquaresCubes n = [n^2, n^3]
```

```
allSquaresCubes :: [Integer] → [Integer]
allSquaresCubes ns = ns >>= calcSquaresCubes.
```

These examples help to understand and appreciate the simplicity and utility of the monad implementation in Haskell. The idea can be used to build larger data pipelines that can become very complex. It should not be misunderstood, though, that monads are simply a design pattern and can be applied in many generic situations.

To compare the definition of a monad in Haskell to a monad in Category Theory, recall the definition of a monad. If `C` is a category, then a monad consists of an endofunctor `T: C → C` and two natural transformations `n: id_C → T` and `u: TT → T`, each of which satisfy the following properties: `u(Tu) = uu` and `un = u(Tn) = id_T`. Consider the definition of the type constructor for a functor:

```
class Functor f where
  return :: a → f a
  fmap :: (a → b) → f a → f b
  join :: f (f a) → f a

  join x = x >>= id

class Monad m => Functor m where
  (>>=) :: m a → (a → m b) → m b

  f >>= x = join $ fmap f x.
```

The function `join` is useful to bring an element out of nested wrappers of type classes. And the function `fmap` is analogous to the definition of the application of a functor to a morphism, as it is applied to the objects over which the morphism is mapped. The bind function is now written in terms of `join` and `fmap`. Utilizing this definition of the type constructor, the monad properties representing associativity and identity can be satisfied. First, the function `join` can be equated with the natural transformation `u: TT → T`, and the function `return` can be equated with the natural transformation `n: id_C → T`. These associations make sense because the first natural transformation combines two functors into one, equivalent to the type of `join` in the `Functor f` type constructor. The function `return` moves an element into the container type class represented in the type constructor, which is equivalent to the natural transformation that maps an object to a functor. Therefore, since these functions are equivalent to the natural transformations, the two monad properties are satisfied by functions defined in the `Functor f` type constructor:

1. `u(Tu) = uu` is equivalent to `join . fmap join = join . join`
2. `un = u(Tn) = id_T` is equivalent to `join . fmap return = join . return = id.`

This shows that, by definition, the concepts of functors and monads in Category Theory parallel those in Haskell. This is intriguing because it is interesting to see difficult concepts being applied to programming design patterns and a useful and simply way. The concepts in Category Theory do not have to be understood in order to program with the design pattern; however, using mathematics as a basis for design patterns lends itself naturally to mathematically proving concepts that might be used in practice, increasing the amount of confidence in the design patterns.

A Little Computability Theory

Computability Theory is a topic in mathematics that includes the study of procedures and their execution. Procedures in Computability Theory are studied in the context of whether they can be executed in a finite number of steps, or, rather, if they halt. A program is defined to halt if it will complete its execution or calculation at some point in time. This type of programmed is called an effective procedure. Therefore, more generally, an effective procedure is defined as a procedure that can be programmed to execute the procedure in a finite amount of time. A set is said to be decidable if, given an object, there exists an effective procedure that can calculate whether or not the object is a member of the set. Examples of decidable sets are usually based on sets of numbers and strings. A trivial example is any set of finite size because, given an object, any finite set can be enumerated and its elements individually checked for equality with the object.

As an example, consider the set of all strings that represent a verifiable mathematical proof. Since each mathematical proof is a procedure that should contain a finite number of sentences, it is an effective procedure. The fact that a proof is verifiable, meaning that any proposed proof of a theorem is either an element or not an element of the set of mathematical proofs, it follows that the set of mathematical proofs is a decidable set. This fact makes Computability Theory central to all foundations of mathematics.

A partial function is defined as a mapping between sets. That is, it is a subset of the Cartesian Product produced from its domain and co-domain. However, it is not a requirement that a partial function be defined for every element of its domain, which is different from that of a function. A partial function is said to be effectively calculable if, given any element in its domain, it eventually halts and returns the correct value for the input provided. If it is not given an element that is an element of its domain, then it does not halt. Therefore, a set is decidable if its characteristic function is effectively calculable. The characteristic function for a set is defined as a function that halts and returns meaningful values based on whether, given any object, the object is a member of the set.

For example, the partial function for subtraction `f(m,n) = m-n if m>=n` is effectively calculable. Another example of an effectively calculable partial function is given by `g(n) = smallest p>n with (p,p+2) being twin primes`. The procedure is to assign `p = n+1` and check whether it is a twin prime. If it is, then halt and return the 2-tuple `(p,p+2)` as output. If it is not, then increment `p` and check. This procedure continues until the program determines twin primes. If the program does not determinate any twin primes, then the program does not halt; however, the function is still effectively calculable because it is not necessary for the program to halt and provide an answer since the number `n` would not have been defined in its domain.

Computability Theory has applications in many areas of theoretical computer science. One notable result that stems in Computability Theory focuses on the halting problem, which is a difficult problem that discusses all computer programs. The problem is stated as follows. Given the universal set of computer programs, does there exist an effective procedure that, given any program and its inputs, can calculate if the program halts. In other words, is it possible to decide whether any program halts, given its inputs?

This question can be answered in terms of specific programming languages or in general. The answer to the question is that the halting problem is undecidable. That is, no algorithm exists that can determine if any program halts, given its input. An easy way to sketch the proof of this claim is to focus on numbers and consider Cantor's proof on infinite sets. Cantor's proof states that there exist infinite sets that are not isomorphic to the set of natural numbers. That is, there exists infinite sets that have different cardinalities. To show this, the first step is to construct any number of non-terminating binary sequences. Then, if these sequences were listed, each sequence could be numbered based on their position in the list. The result of the proof is based on the claim that a new sequence can be constructed that is different from any sequence in the list of sequences. This is done by creating a sequence whose first element is different than the first element of the first list, whose second element is different than the second element of the second list and so forth. Therefore, if the sequence were equal to one of the sequences in the list, say the n th, then its n th element would equal the n th element of the list, which is impossible. This means that the list of all infinite binary sequences can not be enumerated and is not isomorphic to the set of natural numbers. In other words, it is not countable.

Turning back to the the definition of a computable number is a real number of any precision that can be generated using an algorithm that halts. This is equivalent to saying that a real number is computable if there exists an algorithm which computes, given a natural number n, the n th digit of the decimal expansion of the real number. The set of computable numbers is countable due to the fact that a computable number must be given by an algorithm of finite length written in finitely many symbols. Therefore, there are finitely many algorithms in which to count the computable numbers. When Cantor's argument is applied to the list of computable numbers, a new real number is created that is not on the list that. This real number was computed because its construction algorithm consisted of designing a number similar to the construction in Cantor's argument. However, this number must not be in the list of all computable numbers, making it computable and not computable at the same time.

The previous proof sketch is an argument that the halting problem is undecidable because a real number was created that led to an undecidable result. That is, a computable number was formed that the algorithm claimed to be not computable. This is an intriguing proof to study, especially in a more rigorous setting because it brings together concepts between Mathematical Analysis, Set Theory and Computability Theory.

Lambda Functions

Lambda Calculus is a mathematical structure of function definitions that motivates many concepts in functional programming that were not explicitly introduced in the earlier section on functional programming. Many concepts from Lambda Calculus are evident in functional languages such as Haskell. A prime example is lambda functions, which are functions that do not have a name. Lambda functions are also sometimes called anonymous functions.

A function in Haskell that doubles an integer can be written multiple ways. The way to write this function using a named function would be: `double x = 2*x`. The syntax in Haskell to write this as a lambda function is:

`\x → 2*x`.

The slash represents the Greek letter Lambda, and the value that is provided for `x` is sent to the function that multiplies the value by two. Recall that functions are partially applied in Haskell, so this function can also be written simply as: `(2*)`, but the lambda function provides the programmer the ability to explicitly handle the function's parameters without being required name the function. In fact, the type of all of these definitions of the function that doubles an integer is `Integer → Integer`, so any of these functions can be used as a parameter to a higher function that requires a function of this type as input.

In Lambda Calculus, there are variables, abstractions and applications. A variable is a single symbol, such as `x`, that is a parameter. An application is a process that uses the variables to create an output. An abstraction is the expression containing the variables and applications, such as `\x → e`, where `x` is a variable and `e` is an application. It is possible to have multiple variables as well as expressions, such as `\x y → f g`. It is conventional that the application evaluation begins with the left-most application. In this example, the application `f` would be evaluated before the application `g`. A variable is also called free if it does not appear to the right of the lambda symbol before the arrow but it appears in an application to the right of the arrow.

Some examples of functions that can be written as lambda functions include:

1. `\a → a`
2. `\a → a+1`
3. `\a b → a+b`
4. `\a b c d → or $ map (\x → x==0) [a,b,c,d]`
5. `\a b c d → [a,b,c,d] >>= calcSquaresCubes`.

These functions all expect integers as input. The first function is the identity function. The second function increments the integer by one. The third function returns the sum of two integers. The fourth function takes four integers and returns true if any of the integers equal zero. The fifth function takes four integers, builds a list and binds that list to the function that returns a list of all their squares and cubes using a function defined in the section on monadic programming.

There is an important theorem that connects the ideas of computable functions to existence of certain lambda functions. First, the definition of beta-reduction needs to be defined. Beta-reduction is defined by the process: `(\x → a) b => a[b/x]`, where the symbol `a[b/x]` corresponds to substituting the application `b` for the free occurrences of the variable `x` in the application `a`. Examples of beta-reduction include:

1. `(\x → x) (\y → y) => \x → (\y → y) => \y → y`
2. `(\x → x x) (\y → y) => \x → (\y → y) (\y → y) => (\y → y) (\y → y)`
3. `(\x → x (\x → x)) y => y (\x → x)`.

In the first example, the abstraction `(\y → y)` replaces the variable `x` in the application. Then, since the application no longer contains a variable `x`, the symbol `(\x →)` is meaningless in the application. Therefore, it can be removed from the abstraction. The other examples are similar to the first.

The theorem that connects the set of computable functions to the set of lambda functions relies on beta-reduction. The theorem is: A function `f` between the natural numbers is a computable function if, and only if, there exists a lambda abstraction such that for every pair of natural numbers `(m,n)`, `f(m) = n` if, and only if, the beta-reduction `x => y` exists between the lambda abstractions `x` and `y`, which are the Church numerals corresponding to the natural numbers `m` and `n` under the image of `f`, respectively. A Church numeral is just a continuous application of a function. The generic zeroth Church numeral is defined as `\x f → x`. The first Church numeral is defined as `\x f → f x`. The second Church numeral is defined as `\x f → f (f x)`. Therefore, the lambda abstraction `x` represents the application of `f` to `m`, `m` times.

This brief introduction to the formalization of lambda functions is beneficial because lambda functions are an important aspect of functional programming. In functional programming, a function is treated with a type just as a variable. This makes it possible to provide functions as parameters and outputs of other functions, called higher-order functions. Lambda functions provide the programmer with the ability to reason about the flow of data containing functions without providing specific names for those functions. Lambda functions also provide a way of defining functions while explicitly controlling the behavior of the parameters.

Conclusion

Functional programming is a style in the declarative programming paradigm that focuses on the data types in a program as well as the functions that are defined over those data types. This is different from object-oriented programming in that object-oriented analysis focuses on the idea of the abstract objects involved in solving a problem as well as encapsulating their behaviors and states.

Many concepts from functional programming have been discussed, including the idea of type classes and type constructors, functional data structures, Category Theory, the zipper data structure and its connections to Category Theory, monadic programming and its connections to Category Theory, Computability Theory as well as lambda abstractions in the context of Lambda Calculus and Haskell. The study of some of these mathematical concepts is beneficial because Haskell and functional programming, in general, is a mathematical style of programming. Therefore, understanding the mathematics behind the topics in functional programming provides the possibility to utilize further complex methods of abstractions in programming that might not be available otherwise.

Bibliography

Awodey, Steve. (2010). *Category Theory*. New Yrok: Oxford University Press.

Cooper, S. B. (2004). *Computability Theory*. Chapman & Hall/CRC.

Cutland, N. (1980). *Computability, An introduction to recursive function theory.* United Kingdom: Cambridge University Press.

Haskell Applicative Functors. (2010). Retrieved from http://en.wikibooks.org/Haskell/Applicative_Functors.

Haskell Monads. (2011). Retrieved from http://en.wikbooks.org/wiki/Haskell/Monads.

Haskell Zippers. (2011). Retrieved from http://en.wikibooks.org/wiki/Haskell/Zippers.

Huet Gerard. (1997). *Functional Pearl: The Zipper*. United Kingdom: Cambridge University Press.

Oosten, J. (2002). *Basic Category Theory*. The Neterlands: Ytrecht University.

Thompson, Simon. (1999). *Haskell: the craft of functional programming*. Massachusetts: Addison Wesley.

Wadler, Philip. *Monads for functional programming*. University of Glasgow.