THESIS

FUNCTIONAL PROGRAMMING APPLIED TO COMPUTATIONAL ALGEBRA

Submitted by

Ian H. Kessler

Department of Mathematics

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2018

ABSTRACT

FUNCTIONAL PROGRAMMING APPLIED TO COMPUTATIONAL ALGEBRA

Underlying many, if not all, areas of mathematics is category theory, an alternative to set theory as a foundation that formalizes mathematical structures and relations between them. These relations abstract the idea of a function, an abstraction used throughout mathematics as well as throughout programming. However, there is a disparity between the definition of a function used in mathematics from that used in mainstream programming. For mathematicians to utilize the power of programming to advance their mathematics, there is a demand for a paradigm of programming that uses mathematical functions, as well as the mathematical categories that support them, as the basic building blocks, enabling programs to be built by clever mathematics. This paradigm is functional programming. We wish to use functional programming to represent our mathematical structures, especially those used in computational algebra.

ACKNOWLEDGEMENTS

I want to thank God for the opportunity to go to graduate school, and for giving me the strength to persevere through it. I want to thank my mom, dad, and my brothers, Jerry and Joel, for believing in me. I am grateful for my mom for giving me emotional support and words of encouragement throughout my time in graduate school. I thank my dad for giving me tough love and words of wisdom for my development as a professional and as a person. I love you all very much.

I am very grateful for my friend, Gabriela Maria Hernandez, for being a friend that I could relate with throughout this journey. You are a friend that I could always open up to about graduate school. Also, I am really grateful for you helping me get out and enjoy the culture of Fort Collins. You definitely added spice to my life here. I am grateful for my friend, Adam Afandi. I love your sense of humor and I am greatly thankful for helping me keep a good perspective and helping me see the bigger picture throughout my time here. I really enjoyed going out for brunch on Saturdays with you and your wife, Rebecca. You are a really good friend. I am thankful for my academic brother, Brady Tyburski. Your enthusiastic attitude really helped to balance my sarcasm. I really appreciate our differences and how we encouraged one another throughout graduate school. For Gabby, Adam, and Brady, I remember in orientation how we all had fun spending time with each other, giving me a sense of belonging, which gave me assurance that Colorado State University is where I was supposed to go. Everyone needs friends to get through and to enjoy this journey, especially during a time as difficult and memorable as graduate school.

I am grateful for my abstract algebra professor, Dr. Rachel Pries, during my first year in graduate school. You helped to see my potential and encouraged me greatly during my first year. You were also a very patient instructor.

As I am writing this at Linden Street Cafe, I want to thank the people of Linden Street Cafe for creating a welcoming space for work, good conversation, and caffeine. I have really enjoyed walking through Old Town to come to one of the greatest coffee shops here in Fort Collins. Thankyou very much.

Last but not least, I want to thank my advisor, Dr. James Wilson. I really appreciate your sage advice throughout my time here. Thankyou for believing in me and for seeing what I needed to succeed in graduate school. Thankyou for leading me into a good career path, and for helping me to grow as a professional. I have come to really appreciate your “no nonsense" intensity and enthusiasm for mathematics. You are exactly what I needed in an advisor. I hope to stay in touch.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Underlying many, if not all, areas of mathematics is category theory, an alternative to set theory as a foundation that formalizes mathematical structures and relations between them. These relations abstract the idea of a function, an abstraction used throughout mathematics as well as throughout programming. However, there is a disparity between the definition of a function used in mathematics from that used in mainstream programming. For mathematicians to utilize the power of programming to advance their mathematics, there is a demand for a paradigm of programming that uses mathematical functions, as well as the mathematical categories that support them, as the basic building blocks, enabling programs to be built by clever mathematics. This paradigm is functional programming. We wish to use functional programming to represent our mathematical structures, especially those used in computational algebra.

Functional programming is programming with pure functions, objects that only return a unique output for each input, as used in mathematics. With this restriction, our programs become more predictable, and we can treat our functions as first class objects that are easy to glue together by composition and by higher-order functions defined on clever data structures. With our programs built by pure functions as our basic building blocks, there is an increased support for modularity, the ability to break a large program into smaller independent, interchangeable parts, making debugging and editing code more efficient and allowing code to be more concise.

By having our programs more predictable and composable, we are interested in the laws of our operations defined on types, the construction of data structures that implement these operations, and the relationship between these structures. That is, we want to understand the algebra of our structures. Thus, we want to understand the category theory behind our programs, the abstract algebra of composable entities. We will see how algebraic laws and modularity can provide flexibility in methods of evaluation for computations, enabling more efficient–in particular, concurrent–methods to compute while retaining correctness.

Also, by using the ideas of category theory within functional programming, we want to be able to translate the objects and problems in computational algebra into the language of category theory that can be understood by our functional programming language-of-choice, Scala, enabling users to make use of tools from both worlds. We will see how to implement these structures by the use of type classes. We will also see the shortcomings of the desired implementations of mathematical categories with that of the categories interpreted in the language.

In particular, we will look into the category of G-sets as well as the restriction/induction adjoint functor pair that could potentially translate the computational group theoretic Schreier-Sims algorithm into a category-theoretic problem. For future work, we would like to construct a Frobenius monad and/or a Frobenius comonad in Scala that would aid in computing the Schreier-Sims algorithm in a functional manner.

Chapter 2

Functional Programming

2.1 What is Functional Programming?

The term “function" in programming is used informally as any set of commands abstracted into a named synonym which can be used anywhere in code. In particular, it makes few, if any, axiomatic requirements about inputs or outputs and is, thus, unrelated to the strict ideas of functions in math. However, **functional programming** is a programming **paradigm** that stresses **pure functions**, functions that only return a unique output for each input, as used in mathematics. With that in mind, the question that follows is, what would a nonpure function be in a programming language? If we were to input a variable, x, to a function, f, how could the output vary? The output can vary depending on what the **state** of the program is at the time of application of the function. The state encompasses all of the factors that are separate from input that can influence the outcome of your program.

Consider if you are on an elevator and you can either press $\uparrow$ to go one floor up or press $\downarrow$ to go one floor down. Thus, we can think of it like a function that accepts $\uparrow$ or $\downarrow$ as an argument and returns the floor that it transports you to. However, the floor that it transports you to depends on the current floor you are on right now. Also, if you are already on the top floor, then $\uparrow$ will have you remain on the top floor, and, if you are on the bottom floor, then $\downarrow$ will have you remain on the bottom floor. Thus, the elevator function is not a pure function, since it depends on external state.

Here is another example of a nonpure function. Consider the code written in Python:

```
def append7(L):
  L.append(7)
  return L
```

This is a function that accepts a list as an argument and returns the list with 7 appended to it. However, it does not only return a new list. It also changes (mutates) the original list. This additional effect is called a **side effect** of the program. Notice that if I assign `L = [1,2,3]`, then evaluating `append7(L)` returns `[1,2,3,7]`. However, if I evaluate `append7(L)` again, the program returns `[1,2,3,7,7]`, because `L == [1,2,3,7]` at the time of the second evaluation. We see that `append7(L)` does not return the same result in all places it could be evaluated, so it is not regarded as a pure function. It depends on state that can change at the time of evaluation, and the state changes by changing the original list upon evaluation. Another example of changing state are reassigning variables. Thus, functions that rely on variables that can be reassigned or reassign variables are also impure. Overall, pure functions do not depend on state that can change and do not change state. Thus, the values and references that can be passed to pure functions are, by definition, **immutable**, a property that means unchangeable.

2.2 Advantages of Functional Programming

One advantage of pure functions is **data sharing**, which relies on immutable data structures rather than mutable data structures. Looking at the example from above, a list in Python is mutable because there are methods defined on lists that change the data. Other ways of mutating lists in Python are by reassigning different values to the entries of a list. For example, we can assign `L = [1,2,3,4]` and then reassign `L[0] = 7`, which would mutate `L` to `[7,2,3,4]`. Pure functions only accept immutable objects and return new objects, rather than mutating old ones. This uses more memory unless cleverly designed, but it has the advantage of giving **predictable** outcomes. Also, it allows you to reuse objects for other functions, without needing to explicitly copy objects, thereby reducing code. According to Chiusano and Bjarnason [1]“We find that in the large, FP can often achieve greater efficiency than approaches that rely on side effects, due to much greater sharing of data and computation." An example of a list method in Scala that relies on data sharing is `def drop(n: Int)`, which returns a new list that has dropped the first n elements and leaves the original list unchanged. Unless otherwise stated, we will use Scala as our programming language-of-choice for examples of functional programming. Although it is not considered a pure functional language, like Haskell, it greatly supports the functional programming paradigm as well as supports the **object-oriented** paradigm. The power of the Scala language is in the intersection of these paradigms.

Another advantage is the support for **modularity**, the ability to break a large program into smaller independent, interchangeable functions. Although any programming language can make use of modularity, functional programming has better support for it. One reason is that the functions that make up the program do not depend on external state that is carried through the program, so they can be tested independent from the whole. This is beneficial for **debugging** by having the freedom to test individual pieces of code rather than going through the whole program to find bugs. Another reason for the increased support for modularity is the increased ability to glue functions together. In functional programming languages, not only are functions composed together, but they can also be applied to other functions and be returned from functions. In other words, **functions can be treated as data**, enabling programmers to use **higher-order functions (HOFs)**. You can also say that functions are **first class objects**.

Using functions as first class objects and making use of clever data structures helps to condense code considerably and allows for **versatility**. For example, suppose we are given a list of integers, and we want to return a list of integers such that each value has been increased by one. Let us compare two ways we could write this. Consider the following code.

```
def addoneLoop(L: List[Int]): List[Int] = {
  val newlist = collection.mutable.ListBuffer[Int]()
  for (i <- L) {
    newlist += (i + 1)
  }
  newlist.toList
}


def addoneMap(L: List[Int]): List[Int] = L.map(i => i + 1)
```

In the first example, we explicitly create an empty mutable list, write a forloop to add each element to the mutable list, and then make it immutable. However, since Scala makes use of object-oriented and functional programming, we can input the function $i => i + 1$ into the map method of a list object and return a new list such that each entry has been **applied** to the function. We see that the ability to **use functions as arguments** of functions and clever data structures simplify code, minimizing time to write code and making code more readable and correct.

Now, suppose we wanted to create a function that accepts a list of integers as an argument and return a new list where each entry has been multiplied by two. The following code shows this.

```
def multiplytwoMap(L: List[Int]): List[Int] =
     L.map(i => 2 * i)
```

We see a similarity between addoneMap and multiplytwoMap. What we can do is abstract out the commonalities between the two. We can replace both with a more general function that works for all `f:Int=>Int` and all `L:List[Int]`.

```
def fMap(f: Int => Int, L: List[Int]) = L.map(f)
```

The above function is an abstraction that allows for versatility, by having two arguments, one being a function. Suppose we want to partially apply the function. We may want an object that can be applied to a function `f:Int=>Int` in one area of our program and is applied to various lists in other parts of the program. What we can do is **curry** the above function.

```
def fMap(f: Int => Int)(L: List[Int]) = L.map(f)
```

Thus, we can apply `f:Int->Int` to fMap, which returns a function of type `List[Int]->List[Int]` that can be applied to lists of integers. We can **curry** functions because functions can be **returned from functions**.

We see that because of modularity, we can find similarities between different segments of code within the context of a larger program and generalize these pieces by writing a function that can be **reused** to replace these pieces. When functions can be **factored out** from similarities, the amount of code gets reduced, so it becomes more **readable** and enables programmers to implement changes more **efficiently**. We used a simple example above, but we can use this idea for more complex examples. For example, in computational group theory, code that is heavily reused is the orbit/stabilizer algorithm. It is used in the Schreier-Sims algorithm [2], finding centralizer of subgroups [3], finding hypergraph isomorphisms [4], and many more permutation group applications. Thus, it is good to be able to factor out the orbit/stabilizer code for many different uses and test it independent of its uses to see if it works for various sizes of groups, analyze and improve upon the complexity, etc.

With functions as data, we can have variables be **call-by-name parameters** in functions, rather than **call-by-value**, giving the programmer more control to **separate the description of computation from the evaluation of computation**. With the use of **lazy evaluation**, call-by-name parameters can be evaluated and stored once for later use. This method is called **memoization**. With call-by-name parameters, we can implement infinite streams. For example, we can implement an infinite stream of ones by `val ones:  Stream[Int] = Stream.Cons(1, ones).` This is not circular because the second parameter to `Cons` is a call-by-name parameter. With streams, we can create **corecursive functions**. Recursive functions consume data whereas corecursive functions produce data [1].

Another advantage of using pure functions is the simplification of **parallelism**. Parallelism without pure functions requires **shared mutable state**. Having multiple **threads of computation** mutating state simultaneously can create deadlocks, delays, and errors. Suppose we have thread 1, T_1, attempt to change the current state, S, to state S_1, and we have thread, 2, T_2, attempt to change the current state to state S_2. If T_1 changes the state first, then, T_2 is trying to work on a state that no longer exists, since the state has changed to S_1, so there is a high chance of error. However, with pure functions, I can have multiple threads work out their respective function by taking its assigned input and return its own result without interfering with other threads. As each thread is working out its result, the main thread is combining the results to return the final result. By being able to separate the description of a computation from the evaluation of a computation, the programmer has control on when to **fork** a computation to a thread. Also, the programmer has the freedom to **combine** computations without evaluating.

2.3 The Algebras on Types

Because pure functions do not have side-effects, then they are considered **referentially transparent**. A referentially transparent function `f` has the property of the expression of the evaluation of `f(x)` can be replaced with its result anywhere in the program and vice versa, while leaving the meaning of the program unchanged. Consider the nonpure function in Python, which accepts a list, removes the first element of the list, and returns that element:

```
def popHead(L):
  return L.pop(0)
```

Using `popHead`, we see that

```
L = [1,2,3,4]
i = popHead(L)
print(i + i)
```

will print `2`, whereas

```
L = [1,2,3,4]
print(popHead(L) + popHead(L))
```

will print `3`, since there is a side-effect everytime `popHead` is evaluated.

However, with pure functions, we can always replace the result with the expression. For example, consider the pure function in Scala:

```
def concat(s1: String, s2: String) = s1 ++ s2
```

Now, consider the following code:

```
val s1 = "wah"; val s2 = "hoo"; val s3 = "heehoo"
val s4 = concat(s1, s2)
val s5 = concat(s4, s3)
val s6 = concat(s2, s3)
val s7 = concat(s1, s6)
```

We see that `s5 == s7`. In fact, this is true for any `s1, s2, s3`. However, it is not ideal to communicate this generalization using the code above. Since `concat` is referentially transparent, we can replace the intermediate results with the expressions, so we can say that

```
concat(concat(s1, s2), s3) == concat(s1, concat(s2, s3)).
```

Rewriting `concat` with its definition yields

```
((s1 ++ s2) ++ s3) == (s1 ++ (s2 ++ s3)).
```

We see that this is the familiar associative law used in algebra. This example shows that with referential transparency, we can perform **equational reasoning** to see what **laws** should hold for our functions.

Using the referentially transparency of pure functions, we can take advantage of the **algebras** defined on types. An **algebra** on a type describes the operations defined on the values and the laws that the operations hold. By knowing the laws of our algebra, we may be able to use techniques of implementation to optimize our code. For example, suppose we have a list, `L = List(x1, x2, x3, ..., xn)`, and we want to be able to to take the sum of the elements of the list. One way to do this is by using the `foldLeft` method defined on lists. This is the recursion of summing up integers in a list. Here is the trace of the recursion:

```
List(x1, ..., xn).foldLeft(0)((a: Int, b: Int) => a + b) ==
List(x2, ..., xn).foldLeft(x1)((a: Int, b: Int) => a + b) ==
List(x3, ..., xn).foldLeft(x1 + x2)((a: Int, b: Int) => a + b) ==
  ... ==
Nil.foldLeft(x1 + x2 + ... + xn)((a:Int, b:Int) => a + b) ==
x1 + x2 + ... + xn
```

There is also a `foldRight` method that can be used, which could add up the integers starting from the right. However, we see that `(a:Int, b:Int) => a + b` is a pure function, which uses the binary **associative** operation, $+$. Thus, we have a law for our operation:

```
((a + b) + c) == (a + (b + c))
```

which results in

```
(((a + b) + c) + d) == ((a + b) + (c + d))
```

Thus, rather than adding the integers sequentially, we can run parallel computations, since the order in which we combine elements does not matter! We can compute `(a + b)` and `(c + d)` simultaneously, and, then, add the pieces together to get the final result.

In mathematics, if a set A has an **associative binary operation** $A \times A \to A$ and a **unit** for that operation, then we say that $\langle A, \times \rangle$ is a **monoid**. In programming, if a type A has an associative binary function of type `(A, A)-> A` and a **unit** for that function, then we say that `A` forms a **monoid**. In the above example, we see that integers form a monoid under its binary operation, +, where `0` is the unit. Also, the type `String` forms a monoid since ++ is an associative binary function on strings with the empty list, `""`, as the unit. Because of the associative law on the binary operator, we can make use of parallelism.

2.4 Generic Types

Looking back at the previous code

```
def fMap(f: Int => Int)(L: List[Int]) = L.map(f)
```

we see that we have restriced to working only with integers. However, we can abstract this code even more by having it work, not just for `Int` but for all types. We can write

```
def fMap[A, B](f: A => B)(L: List[A]): List[B] = L.map(f)
```

In this case, `A` and `B` are considered **type variables**. Thus, for any two types `A` and `B`, if we have a function of `f:A->B` and a list `L:List[A]`, then fMap(f)(L) returns a list of type `List[B]`. In this implementation, not only does `fMap` abstract over values, but it also abstracts over types. Thus, `fMap` is considered a **polymorphic function**.

Notice that if `A` and `B` are different types, then `List[A]` and `List[B]` are different types. From here, it is worth mentioning that `List` is not a type. Rather, it is a **type constructor**. Type constructors are also called **generic types**. They can be considered a function on types. For example, for any type `A`, `List[A]` is the returned type from the type constructor `List`.

Chapter 3

Category Theory of Representations

3.1 G-Sets

Since functional programming is programming with pure functions, we will understand the algebra of our programs by delving into category theory, since category theory is considered “the abstract theory of functions." [5]. With this context, a **category** consists of

- Objects: $A, B, C, ...$
- Arrows (or Morphisms): $f, g, h, ...$
- For each arrow, f,
  - $dom(f)$ is an object, which is the **domain** of f
  - $codom(f)$ is an object, which is the **codomain** of f
  - $f : A \to B$ indicates that $A = dom(f)$ and $B = codom(f)$
- Given $f : A \to B$ and $g : B \to C$, $g \circ f : A \to$ is the **composite** of f and g
- Given object A, $1_A : A \to A$ is the **identity arrow** of A
- For all $f : A \to B$, $g : B \to C$, and $h : C \to D$
  - $h \circ (g \circ f) = (h \circ g) \circ f$ (Associativity Law)
  - $f \circ 1_A = f = 1_B \circ f$ (Unit Laws)

We will now look at the category of G-sets and later look into how categories can be implemented into Scala.

Definition (G-set)**.** *Given an algebraic structure, $\langle G, * : G \times G \to G, 1 : () \to G \rangle$, a G-set, Ω_G, is a function $* : \Omega \times G \to \Omega$ such that*

- *For all $g, h \in G$ and for all $x \in \Omega$, $(x * g) * h = x * (gh)$.*

- $x * 1 = x$.

The definition of a G-set implies that G needs a binary operation on itself as well as an identity for that binary operation. Also, we see that if $g, h, k \in G$, $x \in \Omega$, and $* : \Omega \times G \to \Omega$ is a G-set, then $x * g(hk) = (x * g) * (hk) = ((x * g) * h) * k = (x * (gh)) * k = x * (gh)k$. Thus, $g(hk)$ and $(gh)k$ are equivalent as endofunctions on Ω, so we may effectively assume G is associative. Thus, G needs to be a monoid. Let $\mathbf{Set_G}$ have the following properties:

- $\mathbf{Obj}(\mathbf{Set_G})$:= G-sets

- $\mathbf{Hom}(\Omega_G, \Delta_G) := \{\phi : \Omega \to \Delta | (\forall x)(\forall g)[\phi(x)g = \phi(xg)]\}$

$$\begin{tikzcd}
\Omega \times G \arrow[r, "\Omega_G"] \arrow[d, "\phi \times 1_G"] & \Omega \arrow[d, "\phi"] \\
\Delta \times G \arrow[r, "\Delta_G"] & \Delta
\end{tikzcd}$$

Figure 3.1: G-Set

- $g \circ_{Set_G} f$:= Function Composition

- 1_{Ω_G} := Function Identity$_\Omega$

Proposition. $\mathbf{Set_G}$ *is a category.*

Proof. Given $\Omega_G, \Delta_G, \Gamma_G \in \mathbf{Obj}(\mathbf{Set_G})$, let $\phi_1 \in \mathbf{Hom}(\Omega_G, \Delta_G)$ and $\phi_2 \in \mathbf{Hom}(\Delta_G, \Gamma_G)$, so $\phi_1 \in \Omega \to \Delta$ and $\phi_2 \in \Delta \to \Gamma$.

Thus, $\phi_2\phi_1 \in \Omega \to \Gamma$, and

$$(\phi_2\phi_1)(x)g = \phi_2(\phi_1(x))g = \phi_2(\phi_1(x)g) = \phi_2(\phi_1(xg)) = (\phi_2\phi_1)(xg).$$

Thus, $\phi_2\phi_1 \in \mathbf{Hom}(\Omega_G, \Gamma_G)$.

If $\phi_1 \in A \to B$, $\phi_2 \in B \to C$, and $\phi_3 \in C \to D$, then function composition ensures that $\phi_3(\phi_2\phi_1) = (\phi_3\phi_2)\phi_1$ and $\phi_1 1_A = \phi_1 = 1_B\phi_1$. Thus, $\mathbf{Set_G}$ is a category. $\square$

An example of a G-set is the set of symmetries of a hexagon, where $\Omega = \{v_0, v_1, v_2, v_3, v_4, v_5\}$, the vertices of the hexagon where v_i is adjacent to $v_{(i+1) \bmod 6}$ for all $i \in \{0, ..., 5\}$, so G is the dihedral group, D_6, generated by $r = (v_0, v_1, v_2, v_3, v_4, v_5)$ and $s = (v_0, v_3)(v_1, v_4)(v_2, v_5)$, and the action is defined by the inclusion map of D_6 into Sym_Ω. Let $\Delta = \{\{v_0, v_3\}, \{v_1, v_4\}, \{v_2, v_5\}\}$. If $* := \Omega_G$, then define $\# : \Delta \times G \to \Delta$ by $\{x, y\}\#g = \{x * g, y * g\}$. It can be shown that $\Delta_G := \#$ is a G-set. Now, define $\beta : \Omega \to \Delta$ by $x \in \beta(x)$ for each $x \in \Omega$. This is well-defined since the sets of Δ are mutually exclusive. It can be shown that $\beta(x * g) = \beta(x)\#g$ for all $x \in \Omega$ and $g \in G$. Thus, $\beta \in \mathbf{Hom}(\Omega_G, \Delta_G)$. This is because Δ is a block system of the action Ω_G.

3.2 Restriction Functor

In category theory, there exists **functors**, which are functions on categories, so a functor $F : \mathbf{C} \to \mathbf{D}$ sends objects of $\mathbf{C}$ to objects of $\mathbf{D}$ and sends arrows of $\mathbf{C}$ to arrows of $\mathbf{D}$ with the following properties for all $A, B, C \in \mathbf{Obj}(\mathbf{C})$:

1. For all $f : A \to B$, $F(f) : F(A) \to F(B)$.

2. For all $f \in A \to B$ and $g \in B \to C$, $F(g \circ f) = F(g) \circ F(f)$.

3. $F(1_A) = 1_{F(A)}$

Here is an example of a functor mapping an act category to another act category:

Let $H \leq G$. Let Res^G_H be a tuple of mappings defined on the objects and the morphisms of $\mathbf{Set_G}$ such that

- $\text{Res}^G_H(\Omega_G) = \Omega_G \restriction_{\Omega \times H} : \Omega \times H \to \Omega$

- $\text{Res}^G_H(\phi) = \phi$

Proposition. Res^G_H *is a functor, mapping* $\mathbf{Set_G}$ *to* $\mathbf{Set_H}$.

Proof. Let $\phi \in \mathbf{Hom}(\Omega_G, \Delta_G)$. Since $H \subseteq G$, it follows that $(\forall x)(\forall h)[\phi(x)h = \phi(xh)]$. Thus, $\mathrm{Res}_H^G(\phi) = \phi \in \mathbf{Hom}(\mathrm{Res}_H^G(\Omega_G), \mathrm{Res}_H^G(\Delta_G))$. Also $\mathrm{Res}_H^G(1) = 1$ and $\mathrm{Res}_H^G(\phi_2\phi_1) = \phi_2\phi_1 = \mathrm{Res}_H^G(\phi_2)\,\mathrm{Res}_H^G(\phi_1)$, so Res_H^G is a functor, mapping $\mathbf{Set_G}$ to $\mathbf{Set_H}$. $\square$

3.3 Induction Functor

Let

- $H \leq G$ be a group
- $\Omega_H \in \mathbf{Set_H}$
- T be a transversal on $H\backslash G$
- ${}^T\overline{(\circ)} : G \to G$ defined by $\overline{g} = T(Hg)$
- ${}^T\,\mathrm{Ind}_H^G(\Omega_H) : \Omega \rtimes_T H\backslash G \times G \to \Omega \rtimes_T H\backslash G$ defined by

$$(x, Ht)\#g = (x * (tg)({}^T\overline{tg})^{-1}, H^T\overline{tg})$$

- If $\phi \in \mathbf{Hom}(\Omega_H, \Delta_H)$, let $\mathrm{Ind}_H^G(\phi) : \mathrm{Ind}_H^G(\Omega_H) \to \mathrm{Ind}_H^G(\Delta_H)$ be defined by $\mathrm{Ind}_H^G(\phi)(x, Ht) = (\phi(x), Ht)$.

When there is little chance for confusion on what transversal is used, the transversal symbol will be dropped from notation.

Lemma 1. *Let $H \leq K \leq G$. If T is a transversal on $H\backslash G$, S is a transversal on $H\backslash K$, and R is a transversal on $K\backslash G$, then*

- $K^{R}\overline{{}^T\overline{g}h} = K^R\overline{gh}$
- $H^{T}\overline{{}^S\overline{g}h} = H^T\overline{gh}$

Proposition. $\mathrm{Ind}_H^G(\Omega_H) \in \mathbf{Set_G}$.

Proof. We see that

$$(x, Ht)1 = (xt(\bar{t})^{-1}, H\bar{t}) = (xt(t)^{-1}, Ht) = (x, Ht).$$

Also,

$$((x, Ht)g_1)g_2 = \text{(By Function Definition)} \tag{3.1}$$

$$(xtg_1(\overline{tg_1})^{-1}, H\overline{tg_1})g_2 = \text{(By Function Definition)} \tag{3.2}$$

$$((xtg_1(\overline{tg_1})^{-1})\overline{tg_1}g_2(\overline{\overline{tg_1}g_2})^{-1}, H\overline{\overline{tg_1}g_2}) = \text{(By Action Property)} \tag{3.3}$$

$$(x(tg_1(\overline{tg_1})^{-1})(\overline{tg_1}g_2(\overline{\overline{tg_1}g_2})^{-1}), H\overline{\overline{tg_1}g_2}) = \text{(By Cancellation)} \tag{3.4}$$

$$(xt(g_1g_2)(\overline{\overline{tg_1}g_2})^{-1}, H\overline{\overline{tg_1}g_2}) = \text{(By Lemma 1)} \tag{3.5}$$

$$(xt(g_1g_2)(\overline{t(g_1g_2)})^{-1}, H\overline{t(g_1g_2)}) = \text{(By Function Definition)} \tag{3.6}$$

$$(x, Ht)(g_1g_2) \tag{3.7}$$

Thus, $\operatorname{Ind}_H^G(\Omega_H) \in \mathbf{Set_G}$. □

Proposition. Ind_H^G *is a functor, mapping* $\mathbf{Set_H}$ *to* $\mathbf{Set_G}$.

Proof. If $\phi \in \mathbf{Hom}(\Omega_H, \Delta_H)$, we see that

$$\operatorname{Ind}_H^G(\phi)(x, Ht)g = \tag{3.8}$$

$$(\phi(x), Ht)g = \tag{3.9}$$

$$(\phi(x)tg(\overline{tg})^{-1}, H\overline{tg}) = \tag{3.10}$$

$$(\phi(xtg(\overline{tg})^{-1}), H\overline{tg}) = \tag{3.11}$$

$$\operatorname{Ind}_H^G(\phi)(xtg(\overline{tg})^{-1}, H\overline{tg}) = \tag{3.12}$$

$$\operatorname{Ind}_H^G(\phi)((x, Ht)g) \tag{3.13}$$

Thus, $\operatorname{Ind}_H^G(\phi) \in \mathbf{Hom}(\operatorname{Ind}(\Omega_H), \operatorname{Ind}(\Delta_H))$.

Furthermore, we see that $\operatorname{Ind}_H^G(1) = 1$. Also, if $\phi \in \mathbf{Hom}(\Omega_H, \Delta_H)$ and $\psi \in \mathbf{Hom}(\Delta_G, \Gamma_G)$, then

$$\operatorname{Ind}_H^G(\psi\phi)(x, Ht) = \tag{3.14}$$

$$((\psi\phi)(x), Ht) = \tag{3.15}$$

$$(\psi(\phi(x)), Ht) = \tag{3.16}$$

$$\operatorname{Ind}_H^G(\psi)(\phi(x), Ht) = \tag{3.17}$$

$$\operatorname{Ind}_H^G(\psi)(\operatorname{Ind}_H^G(\phi)(x, Ht)) = \tag{3.18}$$

$$\operatorname{Ind}_H^G(\psi) \operatorname{Ind}_H^G(\phi)(x, Ht) \tag{3.19}$$

Thus, $\operatorname{Ind}_H^G(\psi\phi) = \operatorname{Ind}_H^G(\psi) \operatorname{Ind}_H^G(\phi)$, so Ind_H^G is a functor, mapping $\mathbf{Set_H}$ to $\mathbf{Set_G}$. □

We see that the construction of $\operatorname{Ind}_H^G(\Omega_H)$ depended on a choice of transversal on $H\backslash G$. We need to show that the functor is equivalent for all transversals, up to isomorphism. To define the equivalence of functors, we need to define arrows between functors. [5] For categories **C**, **D**, and functors $F, G : \mathbf{C} \to \mathbf{D}$, a **natural transformation** $\vartheta : F \to G$ is a family of arrows in **D**: $(\vartheta_C : FC \to GC)_{C \in \mathbf{C}}$ such that for all $f : C \to C'$ in **C**, one has $\vartheta_{C'} \circ F(f) = G(f) \circ \vartheta_C$. A natural isomorphism would exist between F and G, denoted by $F \approxeq G$ if and only if each component $\vartheta_C : FC \to GC$ is an isomorphism.

Suppose T and S are transversals defined on $H\backslash G$.

Proposition. ${}^T\operatorname{Ind}_H^G(\Omega_H) \approxeq {}^S\operatorname{Ind}_H^G(\Omega_H)$.

Proof. Let $\phi_{\Omega_H} \in {}^T\operatorname{Ind}_H^G(\Omega_H) \to {}^S\operatorname{Ind}_H^G(\Omega_H)$ such that $\phi_{\Omega_H}(x, Ht) = (xt({}^S\bar{t})^{-1}, H\bar{t})$. Then,

$$\phi_{\Omega_H}(x, Ht)g = \text{(By Function Definition)} \tag{3.20}$$

$$(xt({}^S\bar{t})^{-1}, H\bar{t})g = \text{(By Action Definition)} \tag{3.21}$$

$$((xt({}^S\bar{t})^{-1})\,{}^S\bar{t}g({}^S\overline{{}^S\bar{t}g})^{-1}, H\overline{{}^S\bar{t}g}) = \text{(By Action Property)} \tag{3.22}$$

$$(xt({}^S\bar{t})^{-1}\,{}^S\bar{t}g({}^S\overline{{}^S\bar{t}g})^{-1}, H\overline{{}^S\bar{t}g}) = \text{(By Cancellation)} \tag{3.23}$$

$$(xtg({}^S\overline{{}^S\bar{t}g})^{-1}, H\overline{{}^S\bar{t}g}) = \text{(By Lemma 1)} \tag{3.24}$$

$$(xtg({}^S\overline{{}^T\overline{tg}})^{-1}, H\overline{{}^T\overline{tg}}) = \text{(Insert Identity)} \tag{3.25}$$

$$(xtg({}^T\overline{tg})^{-1}\,{}^T\overline{tg}({}^S\overline{{}^T\overline{tg}})^{-1}, H\overline{{}^T\overline{tg}}) = \text{(By Action Property)} \tag{3.26}$$

$$((xtg({}^T\overline{tg})^{-1})\,{}^T\overline{tg}({}^S\overline{{}^T\overline{tg}})^{-1}, H\overline{{}^T\overline{tg}}) = \text{(By Function Definition)} \tag{3.27}$$

$$\phi_{\Omega_H}(xtg({}^T\overline{tg})^{-1}, H\overline{tg}) = \text{(By Action Definition)} \tag{3.28}$$

$$\phi_{\Omega_H}((x, Ht)g) \tag{3.29}$$

Thus, $\phi_{\Omega_H} \in \mathbf{Hom}({}^T\mathrm{Ind}_H^G(\Omega_H), {}^S\mathrm{Ind}_H^G(\Omega_H))$. Let $f \in \mathbf{Hom}(\Omega_H, \Delta_H)$. Then

$${}^S\mathrm{Ind}_H^G(f)\phi_{\Omega_H}(x, Ht) = \tag{3.30}$$

$${}^S\mathrm{Ind}_H^G(f)(xt({}^S\bar{t})^{-1}, H\bar{t}) = \tag{3.31}$$

$$(f(xt({}^S\bar{t})^{-1}), H\bar{t}) = \tag{3.32}$$

$$(f(x)t({}^S\bar{t})^{-1}, H\bar{t}) = \tag{3.33}$$

$$\phi_{\Delta_H}(f(x), Ht) = \tag{3.34}$$

$$\phi_{\Delta_H}\,{}^T\mathrm{Ind}_H^G(f)(x, Ht) \tag{3.35}$$

Thus, ${}^S\mathrm{Ind}_H^G(f)\phi_{\Omega_H} = \phi_{\Delta_H}\,{}^T\mathrm{Ind}_H^G(f)$ so $\phi : {}^T\mathrm{Ind}_H^G \Rightarrow {}^S\mathrm{Ind}_H^G$ is a natural transformation.

Let $\psi_{\Omega_H} \in {}^S\mathrm{Ind}_H^G(\Omega_H) \to {}^T\mathrm{Ind}_H^G(\Omega_H)$ such that $\psi_{\Omega_H}(x, Hs) = (xs({}^T\bar{s})^{-1}, H\bar{s})$. Without loss of generality, $\psi_{\Omega_H} \in \mathbf{Hom}({}^S\mathrm{Ind}_H^G(\Omega_H), {}^T\mathrm{Ind}_H^G(\Omega_H))$ and $\psi : {}^S\mathrm{Ind}_H^G \Rightarrow {}^T\mathrm{Ind}_H^G$ is a natural transformation. Furthermore, for all $\Omega_H \in \mathbf{Set_H}$, we have the following:

$$\begin{aligned}
(\psi_{\Omega_H}\phi_{\Omega_H})(x, Ht) &= \text{(By Function Definition)} && (3.36)\\
\psi_{\Omega_H}(xt({}^S\bar{t})^{-1}, H\bar{t}) &= \text{(By Function Definition)} && (3.37)\\
((xt({}^S\bar{t})^{-1})^S\bar{t}({}^T\overline{{}^S\bar{t}})^{-1}, H\overline{{}^S\bar{t}}) &= \text{(By Action Property)} && (3.38)\\
(xt({}^S\bar{t})^{-1S}\bar{t}({}^T\overline{{}^S\bar{t}})^{-1}, H\overline{{}^S\bar{t}}) &= \text{(By Cancellation)} && (3.39)\\
(xt({}^T\overline{{}^S\bar{t}})^{-1}, H\overline{{}^S\bar{t}}) &= \text{(By Lemma 1)} && (3.40)\\
(xt({}^T\bar{t})^{-1}, H\bar{t}) &= ({}^T\bar{t} = t) && (3.41)\\
(xt(t)^{-1}, Ht) &= \text{(By Cancellation)} && (3.42)\\
(x, Ht) & && (3.43)
\end{aligned}$$

Thus, $\psi_{\Omega_H}\phi_{\Omega_H} = 1_{{}^T\operatorname{Ind}_H^G(\Omega_H)}$. Without loss of generality, $\phi_{\Omega_H}\psi_{\Omega_H} = 1_{{}^S\operatorname{Ind}_H^G(\Omega_H)}$ Therefore, for all $\Omega_H \in \mathbf{Set_H}$, ${}^T\operatorname{Ind}_H^G(\Omega_H)$ and ${}^S\operatorname{Ind}_H^G(\Omega_H)$ are isomorphic as objects in the category of $\mathbf{Set_G}$, so ${}^T\operatorname{Ind}_H^G \approxeq {}^S\operatorname{Ind}_H^G$. □

This shows that ${}^T\operatorname{Ind}_H^G \approxeq^S \operatorname{Ind}_H^G$ for any two transversals, T and S, on $H\backslash G$. Thus, the choice of transversal on $H\backslash G$ is arbitrary. We will go back to using the notation, $\overline{(\circ)} = T(Hg)$ and $\operatorname{Ind}_H^G(\Omega_H) : \Omega \rtimes G \to G$ for a fixed transversal, T, on $H\backslash G$.

3.4 Composition of Functors

Proposition. *Let $H \leq K \leq G$ be groups. Then,*

1. $\operatorname{Res}_H^G = \operatorname{Res}_H^K \operatorname{Res}_K^G$
2. $\operatorname{Ind}_H^G \approxeq \operatorname{Ind}_K^G \operatorname{Ind}_H^K$

Proof. 1. ($\operatorname{Res}_H^G = \operatorname{Res}_H^K \operatorname{Res}_K^G$)

We see that $\operatorname{Res}_H^K \operatorname{Res}_K^G(\Omega_G) = \Omega_G \restriction_{\Omega\times K}\restriction_{\Omega\times H} = \Omega_G \restriction_{\Omega\times H} = \operatorname{Res}_H^G(\Omega_G)$. Thus, $\operatorname{Res}_H^G = \operatorname{Res}_H^K \operatorname{Res}_K^G$.

2. ($\operatorname{Ind}_H^G \approxeq \operatorname{Ind}_K^G \operatorname{Ind}_H^K$)

Let $\Omega_H \in \mathbf{Set_H}$. Then, $\operatorname{Ind}_H^G(\Omega_H) : \Omega \rtimes_T H\backslash G \times G \to \Omega \rtimes_T H\backslash G$ and $\operatorname{Ind}_K^G \operatorname{Ind}_H^K(\Omega_H) : \Omega \rtimes_S H\backslash K \rtimes_R K\backslash G \times G \to \Omega \rtimes_S H\backslash K \rtimes_R K\backslash G$ for some transversals T on $H\backslash G$, S on $H\backslash K$, and R on $K\backslash G$.

Let $\phi_{\Omega_H} : \Omega \rtimes_T H\backslash G \to \Omega \rtimes_S H\backslash K \rtimes_R K\backslash G$ be defined by

$$\phi_{\Omega_H}(x, Ht) = (xt({}^R\bar{t})^{-1}({}^S\overline{t({}^R\bar{t})^{-1}})^{-1}, H\overline{t({}^R\bar{t})^{-1}}, K\bar{t}).$$

Looking at the first component,

$$(\phi_{\Omega_H}(x, Ht)g)_1 = \text{(Function Definition)} \tag{3.44}$$

$$t({}^R\bar{t})^{-1}({}^S\overline{t({}^R\bar{t})^{-1}})^{-1}\,{}^S\overline{t({}^R\bar{t})^{-1}}\,{}^R\bar{t}g({}^R\overline{{}^R\bar{t}g})^{-1}({}^S\overline{{}^S\overline{t({}^R\bar{t})^{-1}}\,{}^R\bar{t}g({}^R\overline{{}^R\bar{t}g})^{-1}})^{-1} = \text{(By Cancellation)} \tag{3.45}$$

$$tg({}^R\overline{{}^R\bar{t}g})^{-1}({}^S\overline{{}^S\overline{t({}^R\bar{t})^{-1}}\,{}^R\bar{t}g({}^R\overline{{}^R\bar{t}g})^{-1}})^{-1} = \text{(By Lemma 1)} \tag{3.46}$$

$$tg({}^R\overline{tg})^{-1}({}^S\overline{t({}^R\bar{t})^{-1}\,{}^R\bar{t}g({}^R\overline{tg})^{-1}})^{-1} = \text{(By Cancellation)} \tag{3.47}$$

$$tg({}^R\overline{tg})^{-1}({}^S\overline{tg({}^R\overline{tg})^{-1}})^{-1} = \text{(Insert Identity)} \tag{3.48}$$

$$tg({}^T\overline{tg})^{-1}\,{}^T\overline{tg}({}^R\overline{tg})^{-1}({}^S\overline{tg({}^R\overline{tg})^{-1}})^{-1} = \text{(By Lemma 1)} \tag{3.49}$$

$$tg({}^T\overline{tg})^{-1}\,{}^T\overline{tg}({}^R\overline{{}^T\overline{tg}})^{-1}({}^S\overline{{}^T\overline{tg}({}^R\overline{{}^T\overline{tg}})^{-1}})^{-1} = \text{(Function Definition)} \tag{3.50}$$

$$(\phi_{\Omega_H}((x, Ht)g))_1 \tag{3.51}$$

Looking at the second component,

$$(\phi_{\Omega_H}(x, Ht)g)_2 = \tag{3.52}$$

$$H\overline{\overline{t({}^R\overline{t})^{-1}}{}^R\overline{t}g({}^R\overline{{}^R\overline{t}g})^{-1}} = \tag{3.53}$$

$$H\overline{t({}^R\overline{t})^{-1}{}^R\overline{t}g({}^R\overline{tg})^{-1}} = \tag{3.54}$$

$$H\overline{tg({}^R\overline{tg})^{-1}} = \tag{3.55}$$

$$H\overline{{}^T\overline{tg}({}^R\overline{{}^T\overline{tg}})^{-1}} = \tag{3.56}$$

$$(\phi_{\Omega_H}((x, Ht)g))_2 \tag{3.57}$$

Looking at the third component,

$$(\phi_{\Omega_H}(x, Ht)g)_3 = K\overline{\overline{t}g} = K\overline{tg} = K\overline{{}^T\overline{tg}} = (\phi((x, Ht)g))_3$$

Thus, $\phi_{\Omega_H}(x, Ht)g = \phi_{\Omega_H}((x, Ht)g)$, so $\phi_{\Omega_H} \in \mathbf{Hom}(\operatorname{Ind}_H^G(\Omega_H), \operatorname{Ind}_K^G \operatorname{Ind}_H^K(\Omega_H))$.

Also, if $f \in \mathbf{Hom}(\Omega_H, \Delta_H)$, then

$$\phi_{\Delta_H} \operatorname{Ind}_H^G(f)(x, Ht) = \tag{3.58}$$

$$\phi_{\Delta_H}(f(x), Ht) = \tag{3.59}$$

$$(f(x)t({}^R\overline{t})^{-1}({}^S\overline{t({}^R\overline{t})^{-1}})^{-1}, H\overline{t({}^R\overline{t})^{-1}}, K\overline{t}) = \tag{3.60}$$

$$(f(xt({}^R\overline{t})^{-1}({}^S\overline{t({}^R\overline{t})^{-1}})^{-1}), H\overline{t({}^R\overline{t})^{-1}}, K\overline{t}) = \tag{3.61}$$

$$\operatorname{Ind}_K^G \operatorname{Ind}_H^K(f)(xt({}^R\overline{t})^{-1}({}^S\overline{t({}^R\overline{t})^{-1}})^{-1}, H\overline{t({}^R\overline{t})^{-1}}, K\overline{t}) = \tag{3.62}$$

$$\operatorname{Ind}_K^G \operatorname{Ind}_H^K(f)\phi_{\Omega_H}(x, Ht) \tag{3.63}$$

Thus, $\phi_{\Delta_H} \operatorname{Ind}_H^G(f) = \operatorname{Ind}_K^G \operatorname{Ind}_H^K(f)\phi_{\Omega_H}$, so $\phi : \operatorname{Ind}_H^G \Rightarrow \operatorname{Ind}_K^G \operatorname{Ind}_H^K$ is a natural transformation.

Let $\psi_{\Omega_H} : \Omega \rtimes_S H\backslash K \rtimes_R K\backslash G \to \Omega \rtimes_T H\backslash G$ be defined by
$\psi_{\Omega_H}(x, Hs, Kr) = (xsr({}^T\overline{sr})^{-1}, H\overline{sr})$.

Then,

$$\psi_{\Omega_H}((x, Hs, Kr)g) = \tag{3.64}$$

$$\psi_{\Omega_H}(xsrg({}^{R}\overline{rg})^{-1}({}^{S}\overline{srg({}^{R}\overline{rg})^{-1}})^{-1}, H\overline{srg({}^{R}\overline{rg})^{-1}}, K\overline{rg}) = \tag{3.65}$$

$$(xsrg({}^{R}\overline{rg})^{-1}({}^{S}\overline{srg({}^{R}\overline{rg})^{-1}})^{-1}{}^{S}\overline{srg({}^{R}\overline{rg})^{-1}}{}^{R}\overline{rg}({}^{T}\overline{{}^{S}\overline{srg({}^{R}\overline{rg})^{-1}{}^{R}\overline{rg}}})^{-1}, H\overline{{}^{S}\overline{srg({}^{R}\overline{rg})^{-1}{}^{R}\overline{rg}}}) = \tag{3.66}$$

$$(xsrg({}^{T}\overline{{}^{S}\overline{srg}})^{-1}, H\overline{{}^{S}\overline{srg}}) = \tag{3.67}$$

$$(xsrg({}^{T}\overline{srg})^{-1}, H\overline{srg}) = \tag{3.68}$$

$$(xsrg({}^{T}\overline{{}^{T}\overline{sr}g})^{-1}, H\overline{\overline{sr}g}) = \tag{3.69}$$

$$(xsr({}^{T}\overline{sr})^{-1}{}^{T}\overline{sr}g({}^{T}\overline{{}^{T}\overline{sr}g})^{-1}, H\overline{\overline{sr}g}) = \tag{3.70}$$

$$(xsr({}^{T}\overline{sr})^{-1}, H\overline{sr})g = \tag{3.71}$$

$$\psi_{\Omega_H}(x, Hs, Kr)g \tag{3.72}$$

Therefore, $\psi_{\Omega_H} \in \mathbf{Hom}(\mathrm{Ind}_H^G(\Omega_H), \mathrm{Ind}_K^G \mathrm{Ind}_H^K(\Omega_H))$.

Also, if $f \in \mathbf{Hom}(\Omega_H, \Delta_H)$, then

$$\psi_{\Delta_H} \operatorname{Ind}_K^G \operatorname{Ind}_H^K(f)(x, Hs, Kr) = \tag{3.73}$$

$$\psi_{\Delta_H}(f(x), Hs, Kr) = \tag{3.74}$$

$$(f(x)sr({}^T\overline{sr})^{-1}, H\overline{sr}) = \tag{3.75}$$

$$(f(xsr({}^T\overline{sr})^{-1}), H\overline{sr}) = \tag{3.76}$$

$$\operatorname{Ind}_H^G(f)(xsr({}^T\overline{sr})^{-1}, H\overline{sr}) = \tag{3.77}$$

$$\operatorname{Ind}_H^G \phi_{\Omega_H}(x, Hs, Kr) \tag{3.78}$$

Therefore, $\psi_{\Delta_H} \operatorname{Ind}_K^G \operatorname{Ind}_H^K(f) = \operatorname{Ind}_H^G \phi_{\Omega_H}$, so $\psi : \operatorname{Ind}_K^G \operatorname{Ind}_H^K \Rightarrow \operatorname{Ind}_H^K$ is a natural transformation.

Also, we see that

$$(\psi_{\Omega_H}\phi_{\Omega_H})(x, Ht) = \tag{3.79}$$

$$\psi_{\Omega_H}(\phi_{\Omega_H}(x, Ht)) = \tag{3.80}$$

$$\psi_{\Omega_H}(xt({}^R\overline{t})^{-1}({}^S\overline{t({}^R\overline{t})^{-1}})^{-1}, H\overline{t({}^R\overline{t})^{-1}}, K\overline{t}) = \tag{3.81}$$

$$(xt({}^R\overline{t})^{-1}({}^S\overline{t({}^R\overline{t})^{-1}})^{-1\,S}\overline{t({}^R\overline{t})^{-1}}\,{}^R\overline{t}({}^T\overline{{}^S\overline{t({}^R\overline{t})^{-1}}\,{}^R\overline{t}})^{-1}, H\overline{{}^S\overline{t({}^R\overline{t})^{-1}}\,{}^R\overline{t}}) = \tag{3.82}$$

$$(xt({}^T\overline{{}^S\overline{t}})^{-1}, H\overline{{}^S\overline{t}}) = \tag{3.83}$$

$$(xt({}^T\overline{t})^{-1}, H\overline{t}) = \tag{3.84}$$

$$(xt(t)^{-1}, Ht) = \tag{3.85}$$

$$(xt(t)^{-1}, Ht) = \tag{3.86}$$

$$(x, Ht) \tag{3.87}$$

Also, we see that

$$(\phi_{\Omega_H}\psi_{\Omega_H})(x, Hs, Kr) = \tag{3.88}$$

$$\phi_{\Omega_H}(\psi_{\Omega_H}(x, Hs, Kr)) = \tag{3.89}$$

$$\phi_{\Omega_H}(xsr({}^{T}\overline{sr})^{-1}, H\overline{sr}) = \tag{3.90}$$

$$(xsr({}^{T}\overline{sr})^{-1}{}^{T}\overline{sr}({}^{R}\overline{{}^{T}\overline{sr}})^{-1}({}^{S}\overline{{}^{T}\overline{sr}({}^{R}\overline{{}^{T}\overline{sr}})^{-1}})^{-1}, H\overline{{}^{T}\overline{sr}({}^{R}\overline{{}^{T}\overline{sr}})^{-1}}, K\overline{{}^{T}\overline{sr}}) = \tag{3.91}$$

$$(xsr({}^{R}\overline{{}^{T}\overline{sr}})^{-1}({}^{S}\overline{{}^{T}\overline{sr}({}^{R}\overline{{}^{T}\overline{sr}})^{-1}})^{-1}, H\overline{{}^{T}\overline{sr}({}^{R}\overline{{}^{T}\overline{sr}})^{-1}}, K\overline{{}^{T}\overline{sr}}) = \tag{3.92}$$

$$(xsr({}^{R}\overline{sr})^{-1}({}^{S}\overline{sr({}^{R}\overline{sr})^{-1}})^{-1}, H\overline{sr({}^{R}\overline{sr})^{-1}}, K\overline{sr}) = \tag{3.93}$$

$$(xsr(r)^{-1}({}^{S}\overline{{}^{T}\overline{sr}(r)^{-1}})^{-1}, H\overline{{}^{T}\overline{sr}(r)^{-1}}, Kr) = \tag{3.94}$$

$$(xs({}^{S}\overline{{}^{T}\overline{s}})^{-1}, H\overline{{}^{T}\overline{s}}, Kr) = \tag{3.95}$$

$$(xs({}^{S}\overline{s})^{-1}, H\overline{s}, Kr) = \tag{3.96}$$

$$(xs(s)^{-1}, Hs, Kr) = \tag{3.97}$$

$$(x, Hs, Kr) \tag{3.98}$$

Thus, $\psi_{\Omega_H}\phi_{\Omega_H} = 1_{\mathrm{Ind}_H^G(\Omega_H)}$ and $\phi_{\Omega_H}\psi_{\Omega_H} = 1_{\mathrm{Ind}_H^G(\Omega_H)}$, so $\mathrm{Ind}_H^G(\Omega_H) \approxeq \mathrm{Ind}_K^G \mathrm{Ind}_H^K(\Omega_H)$ for each $\Omega_H \in \mathbf{Set_H}$. Therefore, $\mathrm{Ind}_H^G \approxeq \mathrm{Ind}_K^G \mathrm{Ind}_H^K$. $\square$

3.5 Adjoints

Proposition. *If $H \leq G$, then for all $\Omega_H \in \mathbf{Set_H}$ and for all $\Delta_G \in \mathbf{Set_G}$, there exists a natural isomorphism $\phi : \mathbf{Hom}(\mathrm{Ind}_H^G(\Omega_H), \Delta_G) \approxeq \mathbf{Hom}(\Omega_H, \mathrm{Res}_H^G(\Delta_G)) : \psi$.*

Proof. Let $\Omega_H \in \mathbf{Set_H}$ and $\Delta_G \in \mathbf{Set_G}$. Then, define $\eta_{\Omega_H} : \Omega \to \Omega \rtimes H\backslash G$ by $\eta_{\Omega_H}(x) = (x, H)$. We see that $\eta_{\Omega_H}(xh) = (xh, H) = (x, H)h = \eta_{\Omega_H}(x)h$. Thus,
$\eta_{\Omega_H} \in \mathbf{Hom}(\Omega_H, \mathrm{Res}_H^G \mathrm{Ind}_H^G(\Omega_H))$. Also, if $f \in \mathbf{Hom}(\Omega_H, \Omega'_H)$, then

$$\eta_{\Omega'_H} f(x) = (f(x), H) = \mathrm{Res}_H^G \mathrm{Ind}_H^G(f)(x, H) = \mathrm{Res}_H^G \mathrm{Ind}_H^G(f)\eta_{\Omega_H}(x) \tag{3.99}$$

Thus, $\eta_{\Omega'_H} f = \mathrm{Res}_H^G \mathrm{Ind}_H^G(f)\eta_{\Omega_H}$, so $\eta : 1_{\mathbf{Set_H}} \Rightarrow \mathrm{Res}_H^G \mathrm{Ind}_H^G$ is a natural transformation.

Define $\epsilon_{\Delta_G} : \Delta \rtimes H\backslash G \to \Delta$ by $\epsilon_{\Delta_G}(y, Ht) = yt$. We see that $\epsilon_{\Delta_G}((y, Ht)g) = \epsilon_{\Delta_G}(ytg(\overline{tg})^{-1}, H\overline{tg}) = (ytg(\overline{tg})^{-1})(\overline{tg}) = ytg = \epsilon_{\Delta_G}(y, Ht)g$. Thus, $\epsilon_{\Delta_G} \in \mathbf{Hom}(\operatorname{Ind}_H^G \operatorname{Res}_H^G(\Delta_G), \Delta_G)$. Also, if $f \in \mathbf{Hom}(\Delta_G, \Delta'_G)$, then

$$f\epsilon_{\Delta_G}(y, Ht) = f(yt) = f(y)t = \epsilon_{\Delta'_G}(f(y), Ht) = \epsilon_{\Delta'_G} \operatorname{Ind}_H^G \operatorname{Res}_H^G(f)(y, Ht). \tag{3.100}$$

Thus, $f\epsilon_{\Delta_G} = \epsilon_{\Delta'_G} \operatorname{Ind}_H^G \operatorname{Res}_H^G(f)$, so $\epsilon : \operatorname{Ind}_H^G \operatorname{Res}_H^G \Rightarrow 1_{\mathbf{Set_G}}$ is a natural transformation.

For a fixed $\Omega_H \in \mathbf{Set_H}$ and a fixed $\Delta_G \in \mathbf{Set_G}$, let $\phi : \mathbf{Hom}(\operatorname{Ind}_H^G(\Omega_H), \Delta_G) \to \mathbf{Hom}(\Omega_H, \operatorname{Res}_H^G(\Delta_G))$ be defined by $\phi(f) = \operatorname{Res}_H^G(f)\eta_{\Omega_H}$, and let $\psi : \mathbf{Hom}(\Omega_H, \operatorname{Res}_H^G(\Delta_G)) \to \mathbf{Hom}(\operatorname{Ind}_H^G(\Omega_H), \Delta_G)$ be defined by $\psi(f) = \epsilon_{\Delta_G} \operatorname{Ind}_H^G(f)$.

We see that

$$\phi\psi(f)(x) = \tag{3.101}$$

$$\phi(\epsilon_{\Delta_G} \operatorname{Ind}_H^G(f))(x) = \tag{3.102}$$

$$\operatorname{Res}_H^G(\epsilon_{\Delta_G} \operatorname{Ind}_H^G(f))\eta_{\Omega_H}(x) = \tag{3.103}$$

$$\operatorname{Res}_H^G(\epsilon_{\Delta_G} \operatorname{Ind}_H^G(f))(x, H) = \tag{3.104}$$

$$\epsilon_{\Delta_G} \operatorname{Ind}_H^G(f)(x, H) = \tag{3.105}$$

$$\epsilon_{\Delta_G}(f(x), H) = \tag{3.106}$$

$$f(x), \tag{3.107}$$

and

$$\psi\phi(f)(x, Ht) = \tag{3.108}$$

$$\psi(\operatorname{Res}^G_H(f)\eta_{\Omega_H})(x, Ht) = \tag{3.109}$$

$$\epsilon_{\Delta_G} \operatorname{Ind}^G_H(\operatorname{Res}^G_H(f)\eta_{\Omega_H})(x, Ht) = \tag{3.110}$$

$$\epsilon_{\Delta_G}(\operatorname{Res}^G_H(f)\eta_{\Omega_H}(x), Ht) = \tag{3.111}$$

$$\epsilon_{\Delta_G}(\operatorname{Res}^G_H(f)(x, H), Ht) = \tag{3.112}$$

$$\epsilon_{\Delta_G}(f(x, H), Ht) = \tag{3.113}$$

$$f(x, H)t = \tag{3.114}$$

$$f((x, H)t) = \tag{3.115}$$

$$f(x, Ht). \tag{3.116}$$

Thus, $\phi\psi = 1_{\mathbf{Hom}(\Omega_H, \operatorname{Res}^G_H(\Delta_G))}$ and $\psi\phi = 1_{\mathbf{Hom}(\operatorname{Ind}^G_H(\Omega_H), \Delta_G)}$, so we have an isomorphism $\phi : \mathbf{Hom}(\operatorname{Ind}^G_H(\Omega_H), \Delta_G) \approxeq \mathbf{Hom}(\Omega_H, \operatorname{Res}^G_H(\Delta_G)) : \psi$. This isomorphism is natural in both Ω_H and Δ_G because η and ϵ are natural transformations. □

We call $(\operatorname{Ind}^G_H, \operatorname{Res}^G_H)$ an *adjoint functor pair*, or, in short, an *adjoint*. This generalizes Frobenius Reciprocity, a well-known theorem used in representation theory. We call the natural transformation $\eta : 1_{\mathbf{Set_H}} \to \operatorname{Res}^G_H \operatorname{Ind}^G_H$ the *unit* of the adjoint, and the natural transformation $\epsilon : \operatorname{Ind}^G_H \operatorname{Res}^G_H \to 1_{\mathbf{Set_G}}$ the *counit* of the adjoint. If two functors (F, G) form an adjoint functor pair, then we denote the adjointness by $F \dashv G$.

3.6 Monads

In category theory, a **monad** [5] on a category **C** consists of an endofunctor $T : \mathbf{C} \to \mathbf{C}$, and natural transformations $\eta : 1_{\mathbf{C}} \to T$ and $\mu : T^2 \to T$ satisfying

1. $\mu \circ \mu_T = \mu \circ T\mu$ (Associativity Law)

2. $\mu \circ \eta_T = 1 = \mu \circ T\eta$ (Unit Laws)

Let $T = \operatorname{Res}_H^G \operatorname{Ind}_H^G$, so T is an endofunctor on $\mathbf{Set_H}$, so the unit η is a natural transformation on $\mathbf{Set_H}$ from $1_{\mathbf{Set_H}}$ to T. Also define the natural transformation $\mu : T^2 \to T$ by $\mu_{\Omega_H} = \operatorname{Res}_H^G(\epsilon_{\operatorname{Ind}_H^G(\Omega_H)})$.

Since $\psi(1_{\operatorname{Res}_H^G(\Delta_G)}) = \epsilon_{\Delta_G}$, then $1_{\operatorname{Res}_H^G(\Delta_G)} = \phi(\epsilon_{\Delta_G}) = \operatorname{Res}_H^G(\epsilon_{\Delta_G})\eta_{\operatorname{Res}_H^G(\Delta_G)}$. Suppose $\Delta_G = \operatorname{Ind}_H^G(\Omega_H)$. Then, $1_{\operatorname{Res}_H^G(\operatorname{Ind}_H^G(\Omega_H))} = \operatorname{Res}_H^G(\epsilon_{\operatorname{Ind}_H^G(\Omega_H)})\eta_{\operatorname{Res}_H^G(\operatorname{Ind}_H^G(\Omega_H))}$, so $1_{T(\Omega_H)} = \operatorname{Res}_H^G(\epsilon_{\operatorname{Ind}_H^G(\Omega_H)})\eta_{T(\Omega_H)}$. This gives the following result:

$$1_T = \mu \circ \eta_T$$

Since $\phi(1_{\operatorname{Ind}_H^G(\Omega_H)}) = \eta_{\Omega_H}$, then $1_{\operatorname{Ind}_H^G(\Omega_H)} = \psi(\eta_{\Omega_H}) = \epsilon_{\operatorname{Ind}_H^G(\Omega_H)} \operatorname{Ind}_H^G(\eta_{\Omega_H})$. Thus, $1_{\operatorname{Res}_H^G(\operatorname{Ind}_H^G(\Omega_H))} = \operatorname{Res}_H^G(\operatorname{Ind}_H^G(\eta_{\Omega_H}))\operatorname{Res}_H^G(\epsilon_{\operatorname{Ind}_H^G(\Omega_H)})$, so $1_{T(\Omega_H)} = T(\eta_{\Omega_H})\operatorname{Res}_H^G(\epsilon_{\operatorname{Ind}_H^G(\Omega_H)})$. This gives the following result:

$$1_T = \mu \circ T\eta$$

Let $(((x, Ht), Hs), Hr) \in \Omega \rtimes H\backslash G \rtimes H\backslash G \rtimes H\backslash G$. Then,

$$(\mu_{\Omega_H}\mu_{T(\Omega_H)})((((x, Ht), Hs), Hr)) = \tag{3.117}$$

$$\mu_{\Omega_H}((x, Ht), Hs)r) = \tag{3.118}$$

$$\mu_{\Omega_H}((x, Ht)(sr)(\overline{sr})^{-1}, H\overline{sr}) = \tag{3.119}$$

$$(x, Ht)(sr)(\overline{sr})^{-1}\overline{sr} = \tag{3.120}$$

$$(x, Ht)(sr) = \tag{3.121}$$

$$\mu_{\Omega_H}(((x, Ht)s, Hr)) = \tag{3.122}$$

$$(\mu_{\Omega_H}T(\mu_{\Omega_H})(((x, Ht), Hs), Hr) \tag{3.123}$$

Thus, we have the following:

$$\mu \circ \mu_T = \mu \circ T\mu$$

Therefore, (T, η, μ) forms a monad, where η is the **unit** of the monad and μ is the **join** of the monad.

This is true for any adjoint $F \dashv G$: For $F : \mathbf{C} \leftrightarrow \mathbf{D} : \mathbf{G}$ such that $F \dashv G$ with unit $\eta : 1_{\mathbf{C}} \to GF$ and counit $\epsilon : FG \to 1_{\mathbf{D}}$, then $(GF, \eta, G(\epsilon_F))$ forms a monad [5]. For the monad formed from the $\mathrm{Ind}_H^G \dashv \mathrm{Res}_H^G$ adjoint, we will call it the **Frobenius monad**.

Chapter 4

Category Theory in Programming

How is the abstraction of category theory seen within programming? How is this abstraction useful for programming? Overall, why should we care about this abstraction? These are questions that need to be addressed.

4.1 Monoids

An algebraic structure that appears often in programming is a monoid, which we defined above. Let **Mon** be the category such that

1. **Obj(Mon))** are Monoids
2. $\mathbf{Hom}(\langle M, * \rangle, \langle N, \# \rangle) := \{\phi : M \to N | (\forall (a,b))[\phi(a * b) = \phi(a) \# \phi(b)]\}$
3. $g \circ_{\mathbf{Mon}} f :=$ Function Composition
4. $1_{\langle M, * \rangle} :=$ Function Identity$_M$

Proposition. ***Mon*** *is a category*

Proof. Let $f \in \mathbf{Hom}(\langle M, * \rangle, \langle N, \# \rangle)$ and $g \in \mathbf{Hom}(\langle N, \# \rangle, \langle P, \& \rangle)$. Then, for all $a, b \in M$,

$$(g \circ f)(a * b) = g(f(a * b)) = g(f(a) \# f(b)) = g(f(a)) \& g(f(b)) = (g \circ f)(a) \& (g \circ f)(b)$$

Thus, $g \circ f \in \mathbf{Hom}(\langle M, * \rangle, \langle P, \& \rangle)$. Also, morphism composition is associative since function composition is associative. Thus, **Mon** is a category. $\square$

As we mentioned before, the types `Int` and `String` form monoids under the binary associative operations + and ++, respectively. An example of a monoid homomorphism from `String` to `Int` is the length function:

```
def length(s: String): Int = s.length
```

We see that `length(s1 ++ s2) == length(s1) + length(s2)`, so `length` is a monoid homomorphism.

4.2 TYPE Category

In object-oriented programming (including Scala), we define the existence of classes and objects, where objects are instances of classes. From the class hierarchy, there is a partial ordering between types that is reflexive, transitive, and antisymmetric. Thus, the class hierarchy forms a **poset** (partially-ordered set), and posets are categories. Let us define the category **TYPE** by the following:

1. **Obj(TYPE)** are the types in Scala
2. $\mathbf{Hom}(A, B) = \{A \to B\}$ if and only if $A <: B$ and $\emptyset$ otherwise
3. $(g : B \to C) \circ_{TYPE} (f : A \to B) : A \to C := A \to C$
4. $1_A := A \to A$

This is a category, since $A <: B$ and $B <: C$ implies $A <: C$ by the class hierarchy.

4.3 Entity Category

Let us look at a more general, and more practical, perspective of seeing category theory within functional programming using Scala. Let us define the category **Entity**:

1. **Obj(Entity)**:= Values of All of the Types in Scala
2. $\mathbf{Hom}(a : A, b : B) = \{f : A => B\}$
3. $g \circ_{Entity} f :=$ Function Composition
4. $1_{a:A} :=$ Function Identity$_A$

Proposition. ***Entity*** *is a category.*

If $f \in \mathbf{Hom}(a : A, b : B)$ and $g \in \mathbf{Hom}(b : B, c : C)$, then $g \circ f : A => C$, so $g \circ f \in \mathbf{Hom}(a : A, c : C)$.

4.4 CAT Category

We can also implement our own categories into Scala by use of type classes, which we will discuss later. Here is the type class which we will call **CAT**:

```
trait CAT[C, -->] {
  def Hom(a: C, b: C): Hom[-->]

  def id(a: C): -->

  def compose(g: -->, f: -->): -->

}
```

With this layout, we can implement our own categories and have them fit this description. We also need to define what a `Hom[->]` is, which is a hom-set. This type is a set, but not a set in the programming sense by entering a finite number of values into the set (for example: Set(1,2,3,4)). What we want is for our hom-sets to have a membership test. Here is our implementation of `Hom[->]`:

```
trait Hom[-->] {
  def isArrow(a: -->): Boolean
}
```

Thus, we can check whether a value of the arrow type is an arrow for the category. Later, we will see how we can implement categories. After implementing a category, we can do **property-based testing** to see if is indeed a category. How would we do this? We could have a two generating sets $objects : List[C]$ and $arrows : List[-- >]$. Then, for $a, b \in objects$, we can compute the hom-set, $Hom(a, b)$. Then, we can see find what arrows are in that hom-set. With this pattern, we can find the domains and codomains of our arrows. Then, we can see if composition works by composing and see if the product is in the correct hom-set. We could also see if the unit laws and the associativity laws hold for various arrows. By testing on a sufficiently large list of objects and arrows, we can conclude whether or not a constructed category is indeed a category.

Chapter 5

Implementing Categories in Scala

5.1 Type Classes

We want to implement categories in Scala, and we do this by working with type classes. [7]A **type class** defines some behavior, and a type can join the class by providing that behavior. This is considered to be **ad-hoc polymorphism**, which is less rigid than inheritance. Having a built-in class join a type class is less rigid than trying to inherit from the class. For example, we may want to define an algebra on integers and say `Int` is a subtype of `Monoid`, but `Int` is clearly defined to guarantee Peano's axioms, not to derive its properties from my traits. Thus, since `Int` already has the trait of a monoid, we can use it, but no changes are needed to be made to `Int`. Thus, there is a need for type classes. For the type class `Monoid`, we have the following methods:

```
trait Monoid[A] {
  def op(a1: A, a2: A): A

  def id: A
}
```

We can have a class join the type class by implementing implicit values in the companion object. An **implicit value** is a default value that the compiler can find within its scope.

An example of a class that can join the class `Monoid` is the permutation class. We have a class called `Perm` to implement a permutation, where one of its methods is multiplying another permutation, by function composition, to return a new permutation.

```
class Perm(val act: Int => Int,
    val invAct: Int => Int,
    val degree: Int) {
```

```
    def *(that: Perm): Perm = {
      new Perm(that.act compose this.act,
        this.invAct compose that.invAct,
        this.degree max that.degree)
    }

    ....
}
```

Also, we have a companion object for `Perm` where we have factory methods, one of which is to construct the identity permutation of a given degree, where our default degree is zero:

```
object Perm {
  def id(deg: Int = 0): Perm = new Perm(i => i, i => i, deg)

  ....
}
```

Because we have these methods built into the class and companion object for `Perm`, we can have `Perm` join the class `Monoid` by writing an implicit value in the companion object of `Monoid`:

```
object Monoid {
  implicit val permMonoid = new Monoid[Perm] {
    def op(a1: Perm, a2: Perm): Perm = a1 * a2

    def id: Perm = Perm.id()
  }
  ....
}
```

The following code shows other monoids we can define:

```
object Monoid {
  ....

  implicit val unitMonoid = new Monoid[Unit] {
    def op(a1: Unit, a2: Unit): Unit = ()

    def id: Unit = ()
  }

  implicit val stringMonoid = new Monoid[String] {
    def op(a1: String, a2: String): String a1 ++ a2

    def id: String = ""
  }

  implicit val intMonoid = new Monoid[Int] {
    def op(a1: Int, a2: Int): a1 + a2

    def id: Int = 0
  }

  def functionMonoid[M] = new Monoid[M => M] {
    def op(a1: M => M, a2: M => M): M => M = a2 compose a1
    def id: M => M = m => m
  }
}
```

As we mentioned before, another type class we can define is a category:

```
trait CAT[C, -->] {
  def Hom(a: C, b: C): Hom[-->]

  def id(a: C): -->

  def compose(g: -->, f: -->): -->
}
```

To have our type class `Monoid` have it reflect a category, we will add some more methods:

```
trait Monoid[A] {
  ....

  def gens: List[A]

  def Hom[B](that: Monoid[B]): Hom[A => B] = new Hom[A => B] {
    def isArrow(f: A => B): Boolean = gens.forall(a1 =>
      gens.forall(a2 => f(op(a1, a2)) == that.op(f(a1), f(a2))))
  }
}
```

However, when we try to add an implicit val of a monoid category into the CAT type class, we run into a problem:

```
object CAT {
  implicit val monoidCategory = new CAT[Monoid[?],? => ?] {
   ...
  }
}
```

We see that we run into a problem because `Monoid` is not a type, but a type constructor, and we cannot have a `val` abstract over the type variable of `Monoid`. However, we can create an implicit conversion:

```
object CAT {
  implicit def toMonoid[A] = new CAT[Monoid[A], A => A] {
    def Hom(A1: Monoid[A], A2: Monoid[A]) = A1.Hom(A2)

    def id(A: Monoid[A]): A => A = a => a

    def compose(g: A => A, f: A => A): A => A = g compose f
  }
}
```

We need to give it a type `A` for it to convert to a monoid category, so we created an implicit conversion, which can abstract over types, unlike implicit values.

5.2 Implementing G-Set

Another name for a G-set is an **act**. To write this in Scala, we will first write a class such that our objects from type `M` can be acted on from the left and from the right by values of possibly two different types.

```
class Act[L: Monoid, M, R: Monoid](Lgens: List[L] = Nil,
    Mgens: List[M] = Nil, Rgens: List[R] = Nil)
    (val Lrep: L => M => M, val Rrep: R => M => M) {
```

```
  def Lact(a: L, x: M) = Lrep(a)(x)

  def Ract(x: M, a: R) = Rrep(a)(x)

  ....
}
```

Notice that the type variables `L` and `R` have `Monoid` after each. In this context, `Monoid` is considered a **context bound** for both type variables `L` and `R`. In general, for a polymorphic function that has generic type `T` with a context bound `M`, then there is a requirement that there exists an implicit value of type `M[T]` that needs to be in scope at the place of evaluation of the polymorphic function at compile time. Thus, could `L` and `R` could be of type `Perm`, since we have an implicit value for `Monoid[Perm]`. A natural question that follows is why not just let your generic types `L` and `R` only be type `Perm`? The answer is for versatility. We would like to create a versatile **API** (Application Program Interface) that could potentially be used by a variety of users. There may be users that want to work with permutations as their monoid. Others may want to work with a monoid of matrices. Other users may want to use integers or strings as their monoid, since we saw before that integers and strings form monoids.

We can also create a **left act**, where algebraic objects act on objects only from the left. We can also create a **right act**, where algebraic objects act on objects only from the right. For our G-set defined above, we will mostly focus on the right act.

```
class LAct[L: Monoid, M](Lgens: List[L] = Nil,
    Mgens: List[M] = Nil)((val rep: L => M => M) extends
    Act[L, M, Unit](Lgens, Mgens, Nil)(rep, _ => x => x) {
  def apply(a: L, x: M) = rep(a)(x)

  ....
}
```

```
class RAct[M, R: Monoid](Mgens: List[M] = Nil,
    Rgens: List[R] = Nil)(val rep: R => (M => M)) extends
    Act[Unit, M, R](Nil, Mgens, Rgens)(_ => (x => x), rep) {
  def apply(x: M, a: R) = rep(a)(x)

  ....
}
```

As we proved above, the `RAct` is a category, so we need to implement a Hom method for a given `RAct`, a method that creates a set of arrows (or morphisms) with `this` object and `that` object. Thus, we complete the code for `RAct`:

```
class RAct[M, R: Monoid](Mgens: List[M] = Nil,
    Rgens: List[R] = Nil)(val rep: R => (M => M)) extends
    Act[Unit, M, R](Nil, Mgens, Rgens)(_ => (x => x), rep) {
  def apply(x: M, a: R) = rep(a)(x)

  def Hom(that: RAct[M, R]): Hom[M => M] = new Hom[M => M] {
    def isArrow(a: M => M): Boolean = {
      Rgens.forall(r => Mgens.forall(m => (a(rep(r)(m)) ==
      rep(r)(a(m)))))
    }
  }
}
```

As we said earlier, all a user needs to create a G-set is a type `R` that forms a monoid and a type `M` such that `R` can act on, so we need a representation `rep:R -> M -> M` that gives the properties of a G-set. Thus, this function needs to be a monoid homomorphism between the monoid defined on `R` and the function monoid of type `M -> M`. Once again, we could perform property-based testing to see if the inputted function is a monoid homomorphism, to see if the laws of the G-set hold on our generating lists, `Mgens` and `Rgens`. Since `Monoid` has a `Hom` method, we can implement this:

```
class RAct[M, R: Monoid](Mgens: List[M] = Nil,
    Rgens: List[R] = Nil)(val rep: R => (M => M)) extends
    Act[Unit, M, R](Nil, Mgens, Rgens)(_ => (x => x), rep) {
  def apply(x: M, a: R) = rep(a)(x)

  def Hom(that: RAct[M, R]): Hom[M => M] = new Hom[M => M] {
    def isArrow(a: M => M): Boolean = {
      Rgens.forall(r => Mgens.forall(m => (a(rep(r)(m)) ==
        rep(r)(a(m)))))
    }
  }
  lazy val RActMonoid = new Monoid[R] {
    def op(a1: R, a2: R): R = implicitly[Monoid[R]].op(a1, a2)

    def id: R = implicitly[Monoid[R]].id

    def gens: List[R] = Rgens
  }

  def isRAct: Boolean =
    RActMonoid.Hom(Monoid.functionMonoid[M]).isArrow(rep)
}
```

Similar with `Monoid` and `Category`, there is an issue writing an implicit val for an RAct category::

```
object CAT {
  implicit val RActCategory = new CAT[RAct[?,?], ? => ?] {
    ....
  }
}
```

We cannot create this because `RAct` has type variables. However, we can create an implicit conversion that abstracts over the type variables of `RAct`:

```
object CAT {
  ....

  implicit def toRAct[M, R: Monoid] =
    new CAT[RAct[M,R], M => M] {
      def Hom(a: RAct[M,R], b: RAct[M, R]): Hom[M => M] =
        a.Hom(b)

      def id(a: RAct[M, R]): M => M = m => m

      def compose(g: M => M, f: M => M): M => M = g compose f
    }
}
```

This is the best we can do to show that `RAct` is a category in Scala.

Chapter 6

Functor in FP

6.1 TYPE Functors in Scala

There exists endofunctors defined on the category **TYPE** in Scala. We mentioned before that `List` is a type constructor, a function on types. We can also say that it preserves the hierarchy of the types. For example, since `Int` is a subclass of `AnyVal` in Scala, then `List[Int]` is a subclass of `List[AnyVal]`. This is because the type parameter for `List` is considered **covariant**. In terms of category theory, we can say that there exists an arrow `f:Int->AnyVal`, signifying that `Int<:AnyVal`. Thus, `List(f):List[Int]->List[AnyVal]`, signifying that `List[Int]` is a subclass of `List[AnyVal]`. Thus, the class hierarchy in Scala and other object-oriented programming languages can be seen as a category.

6.2 ENTITY Functors in Scala

We can form the type class for functors, that has an `fMap` method that takes functions of type `A->B` to functions of type `F[A]->F[B]`, so they satisfy the first property shown above. Here is the description for the type class `Functor`:

```
trait Functor[F[_]] {
  def fMap[A, B](f: A => B): F[A] => F[B]
}
```

We can then have `List` and `Option` join the type class `Functor` by writing implicit values for each.

```
object Functor {
  implicit val listFunctor = new Functor[List] {
    def fMap[A, B](f: A => B): List[A] => List[B] =
      L => L.map(f)
  }
  implicit val optionFunctor = new Functor[Option] {
    def fMap[A, B](f: A => B): Option[A] => Option[B] =
      Opt => Opt.map(f)
}
```

We see that there is a correspondence between `map` and `fMap`. The `map` method is defined for objects in the terminal category of the functor, and the `fMap` is a method for the functor itself, so that for function `g:A=>B`, type `F:Functor`, and instance `fa:F[A],Functor[F].fMap(g)(fa) == fa.map(g)`. With the implementation of `fMap`, we can see the law of preserving composition mirrors the identity expressed in category theory: For all `f:A->B` and `g:B->C`, `fMap(g compose f) == fMap(g) compose fMap(f)`.

In programming, **containers** are commonly used to encapsulate data, so that we can look and work on a collection of data values as a whole and to restrict access to them. Although access units of data in a container is restricted, we still want to perform actions on the inside data without explicitly taking out the data. We also want to maintain the structure of the container while being able to act on the data inside. This is where the use of functors comes in.

Let us look back at a previous example:

```
  def addoneLoop(L: List[Int]): List[Int] = {
    val newlist = collection.mutable.ListBuffer[Int]()
    for (i <- L) {
      newlist += (i + 1)
    }
    newlist.toList
  }
  def addoneMap(L: List[Int]): List[Int] = L.map(i => i + 1)
```

Rather than taking out the data, applying a function to each unit of data, and inserting the applied data into the container as shown in `addoneLoop`, we apply a function that is applied to the enclosed data using the `map` method of `List`. We can show that the `List` type constructor fits the description of a functor, since it is a generic type or a type constructor, where the `map` method for `List` maps functions, `f:A->B` to functions on lists–given a list $L : \mathit{List}[A] = \mathit{List}(a_1, a_2, ..., a_n)$ and $f : A \to B$, the result of `L.map(f)` is $\mathit{List}(f(a_1), f(a_2), ..., f(a_n))$. Another way of showing what `map` does is by defining `fMap` method by the following code that uses **pattern matching**:

```
def fMap[A, B](f: A => B)(L: List[A]): List[B] = L match {
  case h::t => f(h)::fMap(f)(t)
  case Nil => Nil
}
```

It can be shown that this implementation of `fMap` for lists is functorial, so List is indeed a functor. From the perspective of our category being values of all types in Scala, our `List` type constructor is an endofunctor, since it maps values to values and functions to functions. There are many other functors in Scala–if there was only one instance of functor, then it would not be worth mentioning. Another example is the `Option` functor where the case classes are `Some(a):Option[A] | None:Option[None]` where the fMap has the following effect:

```
def fMap[A, B](f: A => B)(x: Option[A]): Option[B] = x match {
  case Some(a) => Some(f(a))
  case None => None
}
```

It can be shown that this implementation of `fMap` is also functorial for options. There are many other containers in Scala that can be interpreted as functors. As we said before, with these clever (functor-like) data structures, higher-order methods such as `map` help to simplify code by applying functions to the data inside the containers.

There are also methods that can be interpreted as natural transformations. For example, there is a natural transformation between the `Option` functor and the `List` functor, by means of the `toList` function defined on option values,

```
def toList[A](a: Option[A]): List[A] = a match {
    case Some(a) => List(a)
    case None => Nil
}
```

Notice this works for all parameter types `A`, so we get the **parametricity result** [6]. This result tells us that for all `f:A->B` and `a:Option[A]`, `toList(a.map(f)) == toList(a).map(f)` so our operations commute. Thus, `toList` is indeed a natural transformation from `Option` to `List`. There are also many endofunctors in Scala that have natural transformations which gives the functionality of combining the contained data. The type of functors most commonly used for this are monads.

6.3 Advantages of Laws

There are some advantages of knowing the laws behind our functors. One reason is the ability to create an optimization with the map (or fMap) method. Since the map method preserves composition, then, there can be an optimization in computation, which is known as **map fusion**. Suppose, you have a list of strings, `L`. Then, for each string, you find the length of the string, returning a list of integers. Then, with that list, you want to double each integer in your list, returning again a list of integers. The syntax would be `L.map(i => i.toString).map(i => 2*i)`. However, you are **iterating** through the list twice, when you could iterate through it only once, by `L.map((i => 2*i) compose (i => i.toString))`. By our functor law, the result of these is equal, but the latter is more efficient than the former. If we implement laws into our type class `Functor`, then we can convert one implementation into the other for a more efficient computation. This conversion can be done using pattern matching. Also, as mentioned before, if we have a monoidal category, then we can make use of parallelization (see “Mac Lane’s pentagon" on page 170 of [5]).

In functional programming, we have data that we want to act on and encapsulate to return useful results. Using category theory, we work to understand the algebra of our operations to make our code more predictable to prove we get correct results and to leverage the algebra by using more efficient data manipulation and evaluation techniques that return equivalent (and correct) results.

6.4 Functors in Programming v. Functors in Mathematics

Understand that functors in programming are not equivalent to functors in mathematics. We proved before that $\mathrm{Res}_H^G : \mathbf{Set_G} \to \mathbf{Set_H}$ is a functor. Let us try to write this functor in Scala with the conventions shown above. We see that type constructors that join the type class of `Functor` have methods that include the `map` method like the `List` functor.

```
abstract class List[A] {
  def map[B](f: A => B): List[B] = ...
}
```

The `List` class is abstract because there are two different case classes to implement a list, as shown in previous code, where one case class is the empty list, `Nil`.

Thus, for the `ResGtoH` functor, we will try to write code similar to that of `List.` :

```
abstract class ResGtoH[A] {
  def map[B](f: A => B): ResGtoH[B] = ...
}
```

There is a problem with this signature for the restriction functor. We need it to only accept instances of type `RAct[X, G]` where `X` is a type variable. We do not need to make it abstract, since the only instances of H-sets from the restriction functor are from initializing the class with a G-set. Thus, a more accurate description would be something like the following:

```
type G = ...

class ResGtoH[X](a: RAct[X, G]) {
  def map[Y](f: RAct[X, G] => RAct[Y,G]): ResGtoH[B]
}
```

With this implementation, we see another problem. We need `ResGtoH[X]` to be an H-set for fixed types `H<:G` that also joins the type class `Monoid`. In the `ResGtoH` example, the class constructor cannot be used as a functor between any two categories. Rather, using the class constructor would have the terminal category be itself, not whatever we want it to be. This is because Scala uses **intentional type theory**, whereas mathematics uses **extentional type theory**. Extentional type theory says that things are equal by computational equality. In mathematics, we can say that two functions are equal if their outputs are equal for each input, so their equivalence is determined by their evaluation for all of their arguments. However, the consequence of this in programming would be that type checking is undecidable because programs may not terminate. In intensional type theory, type checking is decidable. However, there needs to be defined equivalences between objects to determine if they are equal. For example, in math, we know that an integer is a rational number, and we know anywhere a rational number can be used in a program, an integer can be used, not changing the meaning of the computations. However, in Scala, there needs to be an implicit conversion from one to the other. Similarly, we need an implicit conversion from `ResGtoH[X]` to `RAct[X,H]`:

```
object ResGtoH {
  type H = ...

  implicit def toRAct[X](r: ResGtoH[X]): RAct[X, H] =
    ...
}
```

Another problem that arises from this implementation of `ResGtoH` (for both the implementation of the class and the companion object) is that this implementation can only work for only one pair of types `H<:G` that can form monoids. If we wanted to create a functor on a different G and H with $H \leq G$, then we need to write a whole new implementation. Thus, we should take a different approach that is more versatile. Consider

```
class Res[G: Monoid, H: Monoid](val incl: H => G) {
  def apply[X](m: RAct[X, G]): RAct[X, H] =
    new RAct()(m.rep compose incl)
  def fMap[X, Y](f: X => Y): X => Y = f
}
```

Rather than using the constructor to map a G-set object to create an object that is implicitly an H-set, we construct the functor using an inclusion map between monoid types that can be applied to any G-set to explicitly create an H-set, so we do not need to write implicit conversions. Since the implementation of `Res` is not a terminal object of the functor, we do not use the `map` method but use the `fMap` which is defined for the functor itself. Above, we defined our arrows for G-sets to be functions between the sets that are acted on, so, for the restriction functor, the arguments for `fMap` are arbitrary functions. Also, we defined the functor's function on arrows to be the identity function, which is shown in the code above. However, the arrow type may be different for different kinds of functors as shown in the type class `CAT`.

6.5 Res/Ind in Scala

We have gone through how to implement the `Res` functor. Here is our implementation:

```
class Res[G: Monoid, H: Monoid](val incl: H => G) {
  def apply[X](m: RAct[X, G]): RAct[X, H] =
    new RAct(m.rep compose incl)
  def fMap[X, Y](f: X => Y): X => Y = f
}
```

We see that we construct the class by a function which tells us what kind of restriction functor it will be. Once the functor is implemented, it works as a function that sends the objects of G-sets to objects of H-sets (by the `apply` method) and sends the arrows of G-sets to arrows of H-sets (by the `fMap` method).

We will now implement the induction functor:

```
class Ind[G: Monoid, H: Monoid](val split: G => (H, G)) {
  def apply[X](m: RAct[X, H]): RAct[(X, G), G] = {
    new RAct((g: G) => (xt: (X, G)) => {
      val (x, t) = xt
      val (h, s) = split(implicitly[Monoid[G]].op(t, g))
      (m(x, h), s)
    })
  }
  def fMap[X, Y](f: X => Y): ((X, G)) => (Y, G) =
    (xt: (X, G)) => (f(xt._1), xt._2)
}
```

For a monoid action, we need to implement how the larger monoid splits into a tuple. In the *Category Theory for Representations* section, we assumed that our monoid was a group, so inverses are defined. With this, we created a `split` map for a fixed transversal for $H\backslash G$, by $g \to (g(\overline{g})^{-1}, \overline{g})$.

Chapter 7

Monads in FP

7.1 Monad Type Class

There is a type class `Monad` used extensively in functional programming. A type that joins the monad–a type that forms a monad–is a type that has an implementation for fMap, unit, and join.

```
trait Monad[T[_]] {
  def fMap[A, B](f: A => B): T[A] => T[B]

  def unit[A](a: A): T[A]

  def join[A](tta: T[T[A]]): T[A]
}
```

The laws for our operations should be for any type `A`,

1. `join(join(x)) == join(fMap(join)(x))` for all `x:T[T[T[A]]]`
2. `join(unit(x)) == x` and `join(fMap(unit)(x)) == x` for all `x:T[A]`

For example, `List` is an example of a monad, where its join method is usually called `flatten`. If I have a list `L = List(List(1,2), List(3,4))`, then `L.flatten` would return `List(1, 2, 3, 4)`. It can be shown that the monad laws for the list implementations of the primitive operations are satisfied:

```
object Monad {
  implicit val listMonad = new Monad[List] {
    def fMap[A, B](f: A => B): List[A] => List[B] = L => L.map(f)
    def unit[A](a: => A): List[A] = List(a)

    def join[A](tta: List[List[A]]): List[A] = tta.flatten
  }
}
```

There are many useful derived operations for `Monad`. A method that is used extensively for lists is `flatMap`. Here are some operations derived from `fMap`, `unit`, and `join`:

```
trait Monad[T[_]] extends Functor[T[_]] {
  def unit[A](a: => A): T[A]

  def join[A](tta: T[T[A]]): T[A]

  def flatMap[A, B](f: A => T[B]): T[A] => T[B] =
    ta => join(fMap(f)(ta))

  def compose[A, B, C](f: A => T[B], g: B => T[C]): A => T[C] =
    a => flatMap(g)(f(a))
}
```

However, in functional programming, rather than fMap, unit, and join as the primitive operations, we can use fMap, unit, and flatMap as the primitive operations. We can implement join in terms of these three by

```
def join[A](ffa: F[F[A]]): F[A] = flatMap(i => i)(ffa)
```

so join can be replaced by flatMap as a primitive operation. In Scala, flatMap is what is used under the hood when you implement **for-comprehensions** in Scala, which are for-loops that construct a collection of values for each iteration. For example, the following code would print `true`.

```
val f: Int => List[Int] = i => List(i,i)
val g: Int => Int = i => 2 * i
val L = List(1,2,3)
val x = for (i <- L; j <- f(i)) yield g(j)
val y = L.flatMap(f).map(g)
println(x == y)
```

In general, if I have a for-comprehension of the form:

```
for(x <- c1; y <- c2; z <- c3) yield {...}
```

Then, that is syntactic sugar for the following:

```
c1.flatMap(x => c2.flatMap(y => c3.map(z => {...})))
```

7.2 Par Monad

7.2.1 Parallel Folding

We will go through how to write a parallel algorithm that folds an indexed sequence. We fold a sequence of elements by combining the data in some way. For a parallel implementation, we will restrict to a type `A` that forms a monoid, and the way we will combine the data is by the binary operation defined for our implicit monoid. As we mentioned before, since the operation is associative, the order of combining elements yields the same answer, allowing us the opportunity to use a concurrent (parallel) implementation.

To construct parallel computations, we will use the library that is built in Chiusano's and Bjarnason's book “Functional Programming in Scala" [1]. One of the major themes described in this book is to “separate the concern of describing a computation from actually running it."

We invent a container type that describes a parallel computation `Par[A]`, where `A` is the type variable that represents the type of the result. We want to be able to control when to `fork` a computation to a separate thread of computation. Thus, we want a `unit` of computation that describes an asynchronous computation done in the current thread and a function that marks a computation for concurrent evaluation by `def fork[A](a => Par[A]): Par[A]`. Since we want to combine values, we need to have a combine method for parallel computations. Thus, we have a trait for `Par`:

```
object Par {
  type Par[A] = ExecutorService => Future[A]

  def unit[A](a:A):Par[A] = (es: ExecutorService) =>
    UnitFuture(a)

  def fork[A](a: => Par[A]): Par[A] = {
    es => es.submit(new Callable[A] {
      def call = a(es).get
    })
  }

  def map2[A,B,C](a:Par[A], b:Par[B])(f: (A,B) => C): Par[C] =
    (es: ExecutorService) => {
      val (af, bf) = (a(es), b(es))
      Map2Future(af, bf, f)
    }

  private case class UnitFuture[A](get: A) extends Future[A] {
    def isDone = true
    def get(timeout: Long, units: TimeUnit) = get
    def isCancelled = false
    def cancel(evenIfRunning: Boolean): Boolean = false
  }
  case class Map2Future[A,B,C](a: Future[A], b:Future[B],
      f: (A,B) => C) extends Future[C] {
    @volatile var cache:Option[C] = None
    def isDone = cache.isDefined
    def isCancelled = a.isCancelled || b.isCancelled
    def cancel(evenIfRunning: Boolean) =
      a.cancel(evenIfRunning) || b.cancel(evenIfRunning)
    def get = compute(Long.MaxValue)
    def get(timeout: Long, units:TimeUnit):C =
      compute(TimeUnit.MILLISECONDS.convert(timeout, units))

    private def compute(timeoutMs: Long): C = cache match {
      case Some(c) => c
      case None => {
        val start = System.currentTimeMillis
        val ar = a.get(timeoutMs, TimeUnit.MILLISECONDS)
        val stop = System.currentTimeMillis
        val at = stop-start
        val br = b.get(timeoutMs - at, TimeUnit.MILLISECONDS)
        val ret = f(ar, br)
        cache = Some(ret)
        ret
      }
    }
  }
}
```

Then, we can fold a sequence by the following function:

```
def fold[A: Monoid](aa: IndexedSeq[A]): Par[A] = {
    if (aa.length <= 1)
      Par.unit(aa.headOption getOrElse implicitly[Monoid[A]].id)
    else {
      val (l, r) = aa.splitAt(aa.length/2)
      Par.map2(fold(l), Par.fork(fold(r)))
      (implicitly[Monoid[A]].op)
    }
  }
```

Then, to run the parallel computation, we apply an executor service to the parallel computation:

```
def run[A](s: ExecutorService)(a:Par[A]):Future[A] = a(s)
```

The return type of this function is a **future**, which is a handle to the computation that allows us to obtain a value and allows us to cancel the computation if necessary.

7.2.2 Par Joins The Monad Type Class

With the above implementation for the function `fork`, there is a chance of deadlocking if we have a fixed thread pool. There is another, more complicated implementation of the `Par` object that does not permit this, which we will not get into. With this implementation, we can define a `join` method:

```
def join[A](a: Par[Par[A]]):Par[A] =
  es => run(es)(run(es)(a).get())
```

With this implementation, then `Par` is a monad.

```
object Monad {
  ....
  implicit val parMonad = new Monad[Par] {
    def fMap[A,B](f: A => B): Par[A] => Par[B} = pa =>
      pa.map2(pa, unit(()))((a,_) => f(a))

    def unit[A](a: => A): Par[A] = Par.unit(a)

    def join[A](tta: Par[Par[A]]): Par[A] = tta.join
  }
}
```

Chapter 8

The Orbit-Stabilizer Algorithm

Suppose I have an element $\alpha \in \Omega$ and a permutation group $G \leq Sym(\Omega)$, that is represented by a set of generators S, so $\langle S \rangle = G$. Then, under the group action $G \times \Omega \to \Omega$ with permutation representation $G \hookrightarrow Sym(\Omega)$, we can find the orbit of α, α^G. What we can also do is for each $\beta \in x^G$, we can find a unique transporter $g \in G$ such that $\alpha^g = \beta$, so we can define a transporter function $U : \alpha^G \to G$, giving a function $\overline{(\circ)} : G \to U(\alpha^G)$. According to the Orbit/Stabilizer theorem [2], this latter function is a transversal for $G_\alpha \backslash G$. With this transversal, we can find the Schreier generators $\{ts(\overline{ts})^{-1} | t \in U(\alpha^G), s \in S\}$, which generate G_α, according to Schreier's lemma [2]. Thus, in finding the orbit, α^G, we can also find generators for G_α. Here are two implementations of the Orbit-Stabilizer algorithm. Here is an implementation in Python:

```
def orbitStab(a, G):
  Omega = [a]
  T = [G.identity()]
  H = []
  for x in Omega:
    for g in G.gens:
      y = g[x]
      if y not in Omega:
        Omega.append(y)
      else:
        h = G.multiply(G.multiply(U[S.index(x)], g), \
        G.invert(U[S.index(y)]))
        H.append(h)
  return (Omega, H)
```

Here is an implementation in Scala:

```
def orbitStab(point: Int, G: PermGroup):(Map[Int, Perm],
    Set[Perm]) = {
  def find(F: Map[Int, Perm])(i: Int, g: Perm):
      Either[Perm, (Int, Perm)] = {
    val j = g(i)
    F.get(j) match {
      case Some(h) => Left(F(i) * g * h.inv)
      case None => Right((j, F(i) * g))
    }
  }
  def go(Delta: Map[Int, Perm], X: Set[Int], S: Set[Perm]):
      (Map[Int, Perm], Set[Perm]) = {
    if (X.isEmpty) { (Delta, S) }
    else {
      val Y = for {i <- X; g <- G.gens } yield find(Delta)(i, g)
      val T = Y.partition(_.isLeft)
      val L = T._1.map(_.left.get)
      val R = T._2.map(_.right.get)
      go(Delta ++ R, R.map(_._1), S ++ L)
    }
  }
  go(Map(point -> G.identity), Set(point), Set[Perm]())
}
```

Chapter 9

Future Work: Frobenius (Co)Monad

We showed that if $H \leq G$ are groups then $\operatorname{Ind}_H^G : \mathbf{Set_H} \leftrightarrow \mathbf{Set_G} : \operatorname{Res}_{\mathbf{H}}^{\mathbf{G}}$ and $\operatorname{Ind}_H^G \dashv \operatorname{Res}_H^G$. Furthermore, we showed that $(\operatorname{Res}_H^G \operatorname{Ind}_H^G, \eta, \operatorname{Res}_H^G(\epsilon_{\operatorname{Ind}_H^G}))$ forms a monad on the category $\mathbf{Set_H}$. However, we may not be interested in the endofunctor action on $\mathbf{Set_H}$ as much as the an endofunctor action on $\mathbf{Set_G}$, since we may be primarily interested in the structure of the larger group G. Thus, let us consider $J = \operatorname{Ind}_H^G \operatorname{Res}_H^G$, so J is an endofunctor on $\mathbf{Set_G}$, so the counit, ϵ is a natural transformation on $\mathbf{Set_G}$ from J to $1_{\mathbf{Set_G}}$.

With our conventions, we have that $\psi = \phi^{-1}$. Since $\phi(1_{\operatorname{Ind}_H^G(\Omega_H)}) = \eta_{\Omega_H}$, then $\psi(\eta_{\Omega_H}) = 1_{\operatorname{Ind}_H^G(\Omega_H)}$. Thus, $\epsilon_{\operatorname{Ind}_H^G(\Omega_H)} \operatorname{Ind}_H^G(\eta_{\Omega_H}) = 1_{\operatorname{Ind}_H^G(\Omega_H)}$ for all $\Omega_H \in \mathbf{Set_H}$. Suppose $\Omega_H = \operatorname{Res}_H^G(\Delta_G)$. Then, we have $\epsilon_{\operatorname{Ind}_H^G(\operatorname{Res}_H^G(\Delta_G))} \operatorname{Ind}_H^G(\eta_{\operatorname{Res}_H^G(\Delta_G)}) = 1_{\operatorname{Ind}_H^G(\Omega_H)}$, so $\epsilon_{J(\Delta_G)} \operatorname{Ind}_H^G(\eta_{\operatorname{Res}_H^G(\Delta_G)}) = 1_{J(\Delta_G)}$.

Since $\psi(1_{\operatorname{Res}_H^G(\Delta_G)}) = \epsilon_{\Delta_G}$, then $\phi(\epsilon_{\Delta_G}) = 1_{\operatorname{Res}_H^G(\Delta_G)}$. Thus, $\operatorname{Res}_H^G(\epsilon_{\Delta_G})\eta_{\operatorname{Res}_H^G(\Delta_G)} = 1_{\operatorname{Res}_H^G(\Delta_G)}$ for all $\Delta_G \in \mathbf{Set_G}$. Therefore,

$$\operatorname{Ind}_H^G(\operatorname{Res}_H^G(\epsilon_{\Delta_G})) \operatorname{Ind}_H^G(\eta_{\operatorname{Res}_H^G(\Delta_G)}) = 1_{\operatorname{Ind}_H^G(\operatorname{Res}_H^G(\Delta_G))},$$

so

$$J(\epsilon_{\Delta_G})) \operatorname{Ind}_H^G(\eta_{\operatorname{Res}_H^G(\Delta_G)}) = 1_{J(\Delta_G)}.$$

From here, we will define the natural transformation $\delta : J \to J^2$ by $\delta_{J(\Delta_G)} = \operatorname{Ind}_H^G(\eta_{\operatorname{Res}_H^G(\Delta_G)})$. We call this transformation the **comultiplication** of the adjoint.

Let $(y, Ht) \in \Delta \rtimes_T H\backslash G$. Then, $\delta_{\Delta_G}(y, Ht) = ((y, H), Ht) \in \Delta \rtimes_T H\backslash G \rtimes_T H\backslash G$. We see that

$$J(\delta_{\Delta_G})((y,H),Ht) = \tag{9.1}$$

$$(\delta_{\Delta_G}(y,H),Ht) = \tag{9.2}$$

$$(\mathrm{Ind}_H^G(\eta_{\mathrm{Res}_H^G(\Delta_G)})(y,H),Ht) = \tag{9.3}$$

$$((\eta_{\mathrm{Res}_H^G(\Delta_G)}(y),H),Ht) = \tag{9.4}$$

$$(((y,H),H),Ht) = \tag{9.5}$$

$$(\eta_{J(\mathrm{Res}_H^G(\Delta_G))}(y,H),Ht) = \tag{9.6}$$

$$\mathrm{Ind}_H^G(\eta_{J(\mathrm{Res}_H^G(\Delta_G))})((y,H),Ht) = \tag{9.7}$$

$$\delta_{J(\Delta_G)}((y,H),Ht) = \tag{9.8}$$

$$\tag{9.9}$$

Therefore, for the endofunctor J and the natural transformations, ϵ and δ, we have the following relations:

$$\delta_J \circ \delta = J(\delta) \circ \delta \text{ (associativity law)} \tag{9.10}$$

$$\epsilon_J \circ \delta = 1_J = J(\epsilon) \circ \delta \text{ (unit laws)} \tag{9.11}$$

Therefore, (J, ϵ, δ) form a **comonad** on $\mathbf{Set_G}$.

Let us see how this relates to the Orbit-Stabilizer algorithm. For the algorithm, we are given a permutation group $G \leq Sym(\Omega)$ by the representation of generators $\langle S \rangle = G$. This defines a G-set for us $\Omega_G : \Omega \times G \to \Omega$ by $x * g = x^g$ where we initially know how each element in S acts on Ω. For a given $\alpha \in \Omega$, we want to find $G_\alpha \leq Sym(\Omega)$ or, in other words, let $H = G_\alpha$ and find the H-set, $\Omega \times H \to \Omega$ generated by the Schreier generators $\{ts({}^T\overline{ts})^{-1} | t \in U(\alpha^G), s \in S\}$ for any transversal T on $G_\alpha \backslash G$.

Notice that by restricting our action Ω_G by Res_H^G and then inducing the action by Ind_H^G, we get a new G-set, $J(\Omega_G) : \Omega \rtimes_T H\backslash G \times G \to \Omega \rtimes_T H\backslash G$ by $(x, Ht)g = (x(tg)({}^T\overline{tg})^{-1}, H^T\overline{tg})$. By the orbit-stabilizer theorem, $T(H\backslash G) = U(\alpha^G)$ for a fixed transversal T and a transporter function U.

Let $\pi_1 : \Omega \rtimes U(\alpha^G) \to \Omega$ be the projection to the first coordinate. Then, we see that $\pi_1 \circ J(\Omega_G) : \Omega \rtimes (U(\alpha^G) \times S) \to \Omega$ gives $x * (t, s) = x(ts)(\overline{ts})^{-1}$, for all $s \in S$ and for all $t \in U(\alpha^G)$, giving a representation of the Schreier generators embedded in $Sym(\Omega)$, therby finding G_α. Thus, we observe a correspondence between acting on a G-set by the comonad $J = \operatorname{Ind}_{G_\alpha}^G \operatorname{Res}_{G_\alpha}^G$ and finding generators for G_α.

This observation shows the possibility of implementing the orbit-stabilizer algorithm in a more functional manner, using a comonadic functor. Similar to monads, we can create a comonad type class:

```
trait Comonad[J[_]] {
  def fMap[A, B](f: A => B): J[A] => J[B]

  def counit[A](ja: J[A]): A

  def comultiply[A](ja: J[A]]): J[J[A]]

  def extend[A, B](f: J[A] => B): J[A] => J[B] =
    ja => fMap(f)(comultiply(ja))

  def compose[A, B, C](f: J[A] => B, g: J[B] => C): J[A] => C =
    ja => g(extend(f)(ja))
}
```

Can we take advantage of the laws of `Comonad` to implement a more efficient orbit-stabilizer algorithm? This seems like a goal worth pursuing.

The orbit-stabilizer algorithm is used within the Schreier-Sims algorithm:

- Given $S \subseteq Sym(\Omega)$
  - Let $G = \langle S \rangle$

- Find B $= \{\beta_1, \beta_2, ..., \beta_k\} \subset \Omega$ such $1 = H_k < H_{k-1} < \cdots < H_1 < H_0 = G$ where $H_i = G_{\beta_1,...,\beta_i}$ for $i \in \{0, ..., k\}$.
- Find T such that
    - $S \subseteq T \subseteq Sym(\Omega)$
    - $\langle T \cap H_i \rangle = H_i$ for $i \in \{0, ..., k\}$

The set B is known as a base for G and T is a strong generator set for (G, B), so (B, T) is known as a **BSGS** for G. Having a BSGS reveals a lot about the structure of the permutation group G. "This representation helps us to calculate the group order, list the group elements, generate random elements, test for group membership and store group elements efficiently." [2] We have an algorithm for the Schreier-Sims algorithm shown in the Appendix. We would like to be able to implement this algorithm in a more functional manner, perhaps using the Frobenius monad and/or the Frobenius comonad.

We know that if $H \leq K \leq G$, then $\operatorname{Res}_H^G = \operatorname{Res}_H^K \operatorname{Res}_K^G$ and $\operatorname{Ind}_H^G \approxeq \operatorname{Ind}_K^G \operatorname{Ind}_H^K$ as shown above. Also, we know the following result:

Proposition. *Suppose $F \dashv G$ such that $\phi : \mathbf{Hom}(F(C), D) \approxeq \mathbf{Hom}(C, G(D)) : \psi$ with $\eta : 1 \Rightarrow GF$ and $\epsilon : FG \Rightarrow 1$ be the unit and counit, respectively for the adjunction, and $\gamma : F \Leftrightarrow H : \theta$. Then, $H \dashv G$ such that $\phi\gamma^* : \mathbf{Hom}(H(C), D) \approxeq \mathbf{Hom}(C, G(D)) : \theta^*\psi$ such that $G(\gamma)\eta$ is the unit and $\epsilon\theta_G$ is the counit.*

Proof. We have that $\phi\gamma^*$ is a natural isomorphism, $\theta^*\psi$ is a natural isomorphism, $(\phi\gamma^*)(\theta^*\psi) = 1$, and $(\theta^*\psi)(\phi\gamma^*) = 1$, so $G \dashv H$.

We see that for each $a \in \mathbf{Hom}(H(C), D)$, we have

$$(\phi\gamma^*)_{C,D}(a) = \phi^*_{C,D}(a\gamma_C) = G(a\gamma_C)\eta_C = G(a)G(\gamma_C)\eta_C.$$

Also, we have for each $b \in \mathbf{Hom}(C, G(D))$, we have

$$(\theta^*\psi)_{C,D}(b) = \theta^*_{C,D}(\epsilon_D F(b)) = (\epsilon_D F(b))\theta_C = \epsilon_D\theta_{G(D)}H(b).$$

Thus, for the adjunction $H \dashv G$, $G(\gamma)\eta$ is the unit and $\epsilon\theta_G$ is the counit. $\square$

Thus, for $H \leq K \leq G$, $\operatorname{Ind}_K^G \operatorname{Ind}_H^K \dashv \operatorname{Res}_H^K \operatorname{Res}_K^G$ where $x \mapsto (x, H, K)$ is the unit, and $(x, Hs, Kr) \mapsto xsr$ is the counit. It can be shown, by induction, if $1 = H_k < H_{k-1} < \cdots < H_1 < H_0 = G$, then $\operatorname{Ind}_{H_1}^{H_0} \operatorname{Ind}_{H_2}^{H_1} \cdots \operatorname{Ind}_{H_k}^{H_{k-1}} \dashv \operatorname{Res}_{H_k}^{H_{k-1}} \operatorname{Res}_{H_{k-2}}^{H_{k-1}} \cdots \operatorname{Res}_{H_1}^{H_0}$ by $x \mapsto (x, H_k, H_{k-1}, \ldots, H_1)$ being the unit and $(x, H_k t_k, H_{k-1} t_{k-1}, \ldots, H_1 t_1) \mapsto x t_k t_{k-1} \cdots t_1$ being the counit.

We would like to explore these ideas more to perhaps implement a functional algorithm to solve this problem, thereby intersecting the world of category theory with the world of computational algebra.

Bibliography

[1] Paul Chiusano, Runar Bjarnason. *Functional Programming in Scala* Manning Publications Co., Shelter Island, NY 11964

[2] Scott H. Murray. *The Schreier-Sims algorithm* Australian National University

[3] William M. Kantor, Eugene M. Luks *Computing in Quotient Groups* p. 18 University of Oregon

[4] Eugene M. Luks *Hypergraph Isomorphism and Structural Equivalence of Boolean Functions* p. 656 University of Oregon

[5] Steven Awodey *Category Theory* Oxford Logic Guides

[6] Philip Wadler *Theorems for Free* University of Glasgow

[7] Cay S. Horstmann *Scala for the Impatient* Addison-Wesley

Appendix

```
def schsims(B: Array[Int] = Array(),
    S: Set[Perm] = Set()): BSGS = {
  def updateGroups(M: Map[Int, PermGroup],
      N: Map[Int, Set[(Int, Perm)]],
      append: Boolean): Map[Int, PermGroup] = {
    def m(k: Int) =
      if (append) M.getOrElse(k, new PermGroup(Set())) else {
        new PermGroup(Set())
      }
    M ++ N.keys.map(k => (k, new PermGroup(N(k).map(_._2)) ++
    m(k)))
  }
  def getLevels(B: Array[Int], T: Boolean => Map[Int, PermGroup],
      U: Map[Int, Map[Int,Perm]], i: Int): BSGS = {
    def stripPlus(L: Array[Int])(g: Perm): (Array[Int],
        Set[PI]) = {
      val (r, d) = strip(B, U, i + 1)(g)
      if (r.isID) { (Array(), Set()) }
      else if (d != L.length) { (Array(), Set((r,d))) }
      else { (Array(r.movedpt().get), Set((r,d))) }
    }
    if (i < 0) { (B, T(true), U) }
    else {
      val (u, h) = T(true)(i).orbitStab(B(i))(U.get(i),
      T(false)(i).gens)
      val (l, y) = extend(B, h.gens, Set())(stripPlus)
      val j = if (y.isEmpty) i - 1 else y.map(_._2).max
      val R = y.flatMap(rd => (i+1 to rd._2).flatMap(d =>
        Set((d, rd._1),
        (d, rd._1.inv)))).groupBy(_._1).updated(i, Set())
      getLevels(l, x => updateGroups(T(x), R, x) ,
      U + (i -> u), j)
    }
  }
  val (b, t) = partialBSGS(this, B, S)
    getLevels(b, x => t, Map(), b.length - 1)
}
```