Arrowized Functional Reactive Programming in Swift using Futures

Matthew Donnelly
B.A.(Mod.) Computer Science
Final Year Project April 2015
Supervisor: Dr. Glenn Strong

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

Matthew Donnelly, April 22, 2015
Permission to lend

I agree that the Library and other agents of the College may lend or copy this report upon request.

Matthew Donnelly, April 22, 2015
Abstract

Functional Reactive Programming is paradigm for modelling time-variant values in a declarative manner. This project presents a framework that applies these concepts to modelling asynchronous code using Futures in Swift. In addition, it answers some questions on writing functional programs using Swift.
Acknowledgements

I’d firstly like to thank my supervisor, Dr. Glenn Strong, for offering his endless support, direction, and advice during this project.

I’d also like to thank my family for the patience, support and encouragement they have given me throughout my degree.
Contents

1 Introduction 3
  1.1 Swift .................................. 3
  1.2 Reactive Systems ........................... 4
  1.3 Goals ........................................ 6
  1.4 Contributions & Structure ...................... 7

2 Background 9
  2.1 Swift .................................. 9
  2.2 Functional Reactive Programming ............ 12
    2.2.1 Motivation ............................. 12
    2.2.2 Classic FRP ............................. 14
    2.2.3 Arrowized FRP ............................ 16
  2.3 Futures & Promises ............................ 23
    2.3.1 Futures ............................... 24
    2.3.2 Promises ............................... 26
  2.4 Summary ..................................... 28

3 Design 29
  3.1 Overview .................................... 29
  3.2 Function Composition .......................... 29
  3.3 Error Handling ................................ 33
  3.4 Summary ..................................... 36

4 Implementation 37
  4.1 Arrows .................................... 37
  4.2 Futures .................................... 41
4.3 Promises ........................................... 47
4.4 Example Program ................................. 48
  4.4.1 API Interaction .............................. 49

5 Results .................................................. 51
  5.1 Conclusions ........................................ 51
  5.2 Limitations ....................................... 52
  5.3 Further Work ..................................... 53
  5.4 Closing Remarks ................................. 55
Chapter 1

Introduction

1.1 Swift

Swift is a new multi-paradigm language created by Apple for Mac and iOS development. It aims to be seen as a more modern and safer replacement for Objective-C. However, despite it largely being a re-imagining of Objective-C, Swift is influenced by a lot of other programming languages, particularly functional ones such as Haskell and Scala.

It can be difficult to precisely define functional programming, as with any programming paradigm. But these are some of the the qualities that well-designed functional programs demonstrate [11, 27, 30]:

Composition: In imperative languages programs are often thought of as a sequence to steps to be executed in order to compute a value. However, in functional programming there is instead an emphasis on writing code than can be repeatedly broken into smaller and smaller expressions. These expressions can be assembled using function application and composition to form a complete program. The focus is on how rather than what. This makes programs easier to reason about and makes code more modular, so it can be reused.

Safe Handling of Mutable State: Object-oriented programming focuses on the design of classes and objects. An object is used to describe a particular “thing” inside a program, each with their own encapsulated state which we manipulate using functions. Programming in this type of a system
using mutable state creates implicit dependencies on all other objects depending on that state. If at any point the state of our program is not what an object expects it to be, our program will not work, so it depends on all other objects to behave as expected. In contrast to this, Functional programming emphasizes the importance of programming with values, free of mutable state or other side effects. Functions do not change any variables or the data of any type outside of the function. By avoiding mutable state, functional programs are safer which facilitates composition and increases modularity. Any interactions with values that require mutable state is usually encapsulated in safe abstraction.

**Well Defined Types:** Functional programs make careful use of types. Types describe the structure of each data value and characterize operations by describing the types of data on which they may be applied to. This allows for more effective compile-time analysis of programs, and makes code simpler, safer and more expressive. It is often the case that well-typed functions used with an expressive type-system can guide the implementation of our programs as we will see later.

It's clear that the properties listed above can be replicated in programs written with Swift, and many believe it already gives us the tools to write functional programs using it [3]. Despite the language still being in the early days of development, there have already been a number of attempts at translating many functional programming paradigms and concepts to Swift. Some of the most notable examples of this are “Swiftz” [29], a repository containing a collection of purely functional data structures and utility functions; and “SwiftyJSON” [26], a library based on the Haskell library Aeson for handling JSON.

To call Swift a functional programming language would be a misnomer, as most of it’s features are still deeply rooted in imperative programming. However, perhaps a better question would be to ask: is it possible to write functional programs in Swift and what benefits does this offers us as programmers?

### 1.2 Reactive Systems

Swift’s primary purpose is for creating applications on Apple’s Mac and iOS platforms. As is the case with many modern programs, those written using Swift
often involve interacting with external entities and asynchronous resources such as web APIs or threads performing computation concurrently. In these types of systems, programs are often expected to react to events that occur during execution. As such, they are referred to as reactive systems. The benefit of writing these kinds of reactive systems is that they allow for programs to interact with the real world. However, this also creates an added layer of complexity in our programs.

The main source of problems when modelling a reactive system is that program execution no longer follows a simple linear path which creates a variety of problems. For example, upon successful completion of one asynchronous procedure you might want to trigger another, creating a chain of asynchronous operations. In Objective-C this is typically handled using nested callbacks (a procedure specified to be invoked on completion of a task). If were to take this approach with Swift it might look something like this:

```swift
TCP.connect(host) { socket, err in
  if err == nil {
    socket.write(message) { err in
      if err == nil {
        socket.read() { responseData, err in
          if err == nil {
            if let parsed = parseResponse(responseData) {
              print(parsed)
            } else {
              handleError(err)
            }
          } else {
            handleError(err)
          }
        } else {
          handleError(err)
        }
      } else {
        handleError(err)
      }
    } else {
      handleError(err)
    }
  } else {
    handleError(err)
  }
}
```

In this example we are opening a TCP connection to a host, sending a mes-
sage to it, reading the response and parsing it. We specify that these actions happen after each other by nesting each one in the callback of the previous action. At each stage in the chain of operations, we need to check if an error has occurred in order to make sure our code is safe, despite the fact that we are handling errors the same way each time. These two elements combined cause a variety problems with our code.

The problem here is that designing, implementing, and maintaining reactive software is difficult! Reactive software systems present several engineering challenges not found in conventional synchronous systems [5]. In particular:

- They are naturally concurrent. Events can arrive asynchronously from multiple sources during execution and frequently trigger more events. Often, they also need to perform continuous processing, such as updating a display of changing data. Coordinating such concurrent computations is difficult and can involve complex synchronization patterns.

- Their state changes over time and there are typically dependencies between data. Hence, when one thing changes, the programmer must update everything that depends on it.

- Their control structure is inverted: instead of the application controlling itself, interaction with external entities determines what computations happen and when. For this reason, reactive systems are often structured around callbacks, functions passed to be executed in response to events.

It’s easy to see how each of these applies to the example above and how easy it is for code to increase in complexity as we chain asynchronous procedures. The primary source of these problems is that we don’t have an effective way of modelling asynchronous behaviour in most imperative programming languages.

1.3 Goals

The main goal of this project was to take concepts from the realm of functional programming and apply them to Swift to solve some of the problems mentioned above. Specifically I was interested in exploring in Functional Reactive Programming and Futures. Functional Reactive programming is paradigm for modelling
time-variant values. Futures are a programming model that deal with deferred results in asynchronous programming.

I aimed to combine these two technologies to create a new way of modelling reactive systems in Swift. This project also serves as a way of testing the expressiveness of Swift’s type system and determining whether it’s possible to write functional programs in it.

1.4 Contributions & Structure

The work presented in this project is greatly inspired by the work of Paul Hudak and Conal Elliott’s paper on Fran [8] as well as Hudak’s paper Arrows, Robots, and Functional Reactive Programming [10]. My motivation was that implementing FRP in a fully expressive manner and in a way that scales efficiently has proved to be a challenging task, even in the world of functional programming. The idea to apply it’s concepts to an imperative language so early in development like Swift, seemed like the perfect way to test both FRP and Swift’s limitations.

The primary contributions presented in this project in terms of code are:

1. A Scala-inspired Futures & Promises implementation

2. An Arrowized FRP framework for Futures

The structure of this report is as follows:

Chapter 2 gives a detailed overview of the primary technologies used in this project. It first outlines some of the important features of Swift used in this project. Following this, it introduces the fundamental concepts of FRP focusing on Arrowized-FRP and describes the differences between some of major implementations. It then outlines the concept of Futures and explains how they are used with particular reference to Scala.

Chapter 3 outlines some of the key design concepts that went into the development of this project.

Chapter 4 explains the implementation of the design features and technologies used in the project. It also gives an example of a program written using it.
Chapter 5 discusses the results of this project and some of the conclusions I have come to subsequently. It also outlines some of the project’s limitations and it’s potential for further work.
Chapter 2

Background

2.1 Swift

The primary focus of this project is on Swift so a degree of familiarity with it is required. In this section I'll be introducing some of the important features used in this project that are not present in most other imperative languages.

Constants, Variables & Types

In Swift, we declare constants with the `let` keyword and variables with the `var` keyword. Swift is a strongly typed language, as such every value must have associated type. This allows the compiler to type check our code to catch and fix certain types of errors. We can choose to explicitly specify the type of a value or allow the compiler to infer it for us. For example:

```swift
let languageName: String = "Swift"
var versionNumber = 1 // inferred as type Int
```

Here we are declaring a constant called `languageName` which we are explicitly assigning a type to, and variable called `versionNumber` which we dont specify a type for but is inferred as type `Int`.

Optionals

Swift also introduces Optional types which allow us to express, at a type level, values that may be absent or computations that may fail. For example we might
want to lookup a value in a dictionary using a key which might not be present. In this case we can make the value returned by dictionary optional to indicate that the value for might not exist. When an optional value is absent, its value is equal to nil. We denote a type as an optional by including a question mark at the end of it e.g. Int?.

Optional Binding is used to test if an optional contains a value, and if so, makes that value available as a temporary constant or variable. It can be used with if and while statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single expression. For example:

```swift
if let constantName = someOptional {
    statements
}
```

**Functions & Closures**

To declare a function in Swift we use the `func` keyword. When declaring one, we can optionally define one or more named, typed values that the function takes as input (parameters), and/or a type of value that the function will deliver as output (return type). For example if we wanted to declare a function called `sayHello`, which takes a name represented by a String as input and returns a greeting for that name, we could write it as follows:

```swift
func sayHello(personName: String) -> String {
    let greeting = "Hello, " + personName
    return greeting
}
```

```swift
let greeting: String = sayHello("John") // "Hello, John"
```

We can write the type of this function as `(String) -> String`. In Swift functions are first-class values so you can use function types just like any other types. For example, you can define a constant or variable to be of a function type and assign an appropriate function to that variable:

```swift
let helloFunction: (String) -> String = sayHello
```

Here we have declared a constant of type `(String) -> String` and assigned it sayHello function as a value. We can also use a function type like this as a parameter type for another function:
func printHelloResult(helloFunction: (String) -> String, name: String) {
    println("Result: "+helloFunction(name))
}

printHelloResult(helloFunction, "John") // prints "Hello, John"

Swift also let us define self-contained blocks of code called Closures that can be passed around and used in your code just like functions. For example, if we wanted to declare one that adds increments a number we could write it as follows:

let incrementor: (Int) -> Int = { a in return a + 1 }

incrementor(1) // returns 2

There are few things here that the compiler can infer for us which we can omit. Firstly, Swift can infer the parameter and return type of closures so we can remove the explicit type annotations. It also lets us use shorthand argument names to refer to parameters which lets us omit the list of parameters as well as the `in` keyword. Instead, we can reference the first argument using `$0`, the second by `$1`, and so on. Finally, for closures which return a value immediately, Swift also let us omit the `return` keyword. We can now rewrite the `incrementor` function as follows:

let incrementor = { $0 + 1 }
incrementor(1) // returns 2

Swift also defines a concept known as trailing closures. When passing a closure as the last argument of a function, you can place that closure outside the parentheses of the function call to allow for cleaner code. For example:

func applyToNumber(number: Int, function: Int -> Int) -> Int {
    return function(number)
}

// returns 2
applyToNumber(1) {
    $0 + 1
}
These are just some the features that Swift offers us [15]. But for the purposes of this project these are the ones we’ll making use of most frequently.

2.2 Functional Reactive Programming

Preface: In this section we examine some advanced functional programming concepts which require some basic knowledge of Haskell. It’s not necessary to understand everything that is covered but the key thing to take away is that Functional Reactive Programming provides an expressive model for time-variant values. The model provided by a specific family of it, called Arrowized FRP, can be applied in other areas to model different types of code [10]. In this project we’ll be using it to model asynchronous code using Futures.

Purely functional programming languages are normally very strict on how programs can interact with external entities [17]. Computations are treated as pure mathematical functions, such that functions always deliver the same outputs given the same inputs. To allow program reliability, this property is mutually dependant on all values being immutable and avoiding changing state. The constraints of purely functional programming mean that programs are protected from implicit dependencies created by mutable state.

However, in many systems important values are those that do change state during execution. Programs are often concerned with actions such as pressing a key or receiving data over a network. How can we model these values which change state in a purely functional way?

In this section we examine Functional Reactive Programming (FRP) and how it allows programs to interact with external entities by modelling values that change state using safe abstractions.

2.2.1 Motivation

Functional Reactive Programming (FRP) is a declarative paradigm for working with reactive systems. We can often consider a program to be a function of it’s inputs (referred to as transformational programming) but unfortunately in
many application domains this model cannot be used. For example, in a concurrent system, computations are performed at overlapping time periods. So, processes need to be synchronized inside the model. In many systems, input is not known in advance, but instead arrives continuously during execution as discrete events, often triggering new events. These are an example of values which change state during execution.

An imperative approach to handling these events often involves repeatedly waiting for an event to occur inside the system and performing some routine based on it. This causes application control structure to be inverted: instead of the application controlling itself, interaction with external entities determines what computations happen and when by triggering events.

In order to prevent this, FRP is instead orientated around the data flow of these events and propagation of change over time. In FRP values that change state are treated as signals, emphasising their time-varying nature. Signals can represent any value which changes over time. For example in the programming language Elm, the signal for the position of the mouse is called `Mouse.position`

1. When the mouse moves, the value of `Mouse.position` changes automatically. By chaining, combining, and reacting to signals, programs can be written in declarative manner which eliminates the need to write code that continually observes and updates values.

Signals can also be transformed and combined like any other value. For instance, if we had a function that takes a mouse position as input and determines whether the mouse’s x-coordinate is an odd number, we can apply this function to `Mouse.position`. This results in a signal of boolean values that updates automatically to true when the mouse’s x-coordinate is odd or false otherwise. Taking this kind of purely functional approach to reactive programming brings with it the typical advantages of functional programming, such as powerful facilities for modularity, composition, abstraction and ease of reasoning [11].

FRP has taken many forms since it was first formulated. Implementations usually differ around discrete vs. continuous semantics and how FRP systems can be changed dynamically. In the following sections we will focus on two of the major families of Functional Reactive Programming: Classic FRP and Arrowized FRP. We will see how the semantics of FRP have evolved and later how they can be applied to Swift.
2.2.2 Classic FRP

Functional Reactive Programming was first devised by Conall Elliott and Paul Hudak in their paper *Functional Reactive Animation*, published in 1997 [8]. Its semantics were originally outlined as a model for an animation library embedded in Haskell called Fran. This model has since come to be known as “Classic FRP”. In their initial implementation they introduced two types of reactive values: Behaviours and Events.

**Behaviours** are time-varying values which are represented as a function from time to a value (what we now call signals).

\[
\text{Behaviour } a = \text{Time} \rightarrow a
\]

The domain of Time is considered to be infinite in FRP. As such, Signals are modelled as functions from Time to a value a, where time is represented as non-negative real number i.e. At time \( t \), the behavior has value \( v \) where \( \{ t \in \mathbb{R}_{\geq 0} \} \). Behaviours can be applied to many kinds of task in animation, particularly those that involve modelling physics.

**Events** are collections of instantaneous values represented as a sequence of time-value pairs. In Haskell, they are represented as a list of time and value pairs:

\[
\text{Event } a = [(\text{Time}, a)]
\]

Events can be used to model occurrences in the real world such as user input or network activity. They can also be used to represent predicates based on animation parameters such as collision detection.

In Classic FRP Behaviors and Events are both first class values. Reactivity is modeled using combinators which take an initial behaviour and an event with a new behavior as occurrence value [8]. The resulting behavior acts as the first behavior up to the first occurrence of the even, and then acts as the behavior encapsulated inside the event. This is performed using an operator usually referred to as `until`.

Here is an example of how it can be used to create a Behaviour of colours that cycles between red and green, each time the left mouse button is pressed:
In this example we have defined two behaviours `firstCycle` and `secondCycle`. The value of the behaviour `firstCycle` is equal to `red` until the `leftMousePress` event occurs. We specify a new behaviour following an event using the `(*)=>` operator defined in Fran, which in this case is used to change to `secondCycle`. Similarly, the value of the `secondCycle` behaviour is equal to `green` until the `leftMousePress` event occurs.

**Signal Graphs**

Classic FRP defines a family of functions in the form `lift_n`, which lift functions of arity `n` defined on static values to analogous functions defined on Behaviours [8]. This is accomplished by defining an operator for each arity in the form:

```
lift_n :: (a_1 -> ... -> a_n -> b) -> Behaviour a_1 -> ... -> Behaviour a_n -> Behaviour b
```

**Note:** The :: identifier can be read as “has type”

This allow us to write functions which take multiple values encapsulated by Signals as input, such as mouse position or window dimensions. Using this, programs can be constructed by composing Events and Behaviours and lifting functions or values to Behaviours. This creates what is commonly referred to as a Signal Graph, referring to the flow of signal values through functions which we can treat as nodes of computation [7]. We can picture this graph as a flow chart where wires represent Signals and boxes represent Signal Functions with one type of Signal entering a box and another type exiting. In many ways this model is no different to chaining a series of nested callbacks, however the key difference is that FRP allows us to express this in a declarative manner. This is what makes it possible to keep our code pure and facilitates composition in programs written using FRP.

Unfortunately, the kind of unrestricted access to signals presented in Classic FRP can make it easy to generate both time and space leaks [19]. This is due to the often unpredictable consumption of space in lazy languages such as Haskell. Space leaks occur when accumulated computation becomes quite
large over time and continues to take up more and more memory. Similarly, time leaks occur in real-time systems when a computation does not keep up with the current time, which requires catching up at a later time.

**First Generation FRP**

Since Classic FRP was first formulated it has inspired various other implementations which attempt to fix some of these issues with efficiency as well as space and time leaks. Most aim to retain the expressiveness of Classic FRP while staying true to its original design. The first family of FRP models that this gave rise to what I refer to as **First Generation FRP** (FG-FRP), which is present in languages such as Elm. The core design principles that members of this family maintain are as follows:

- **Signals are connected to the world** – Signals in our programs are directly connected to external entities such as the mouse or keyboard rather than internal ones
- **Signals are infinite** – Inputs to our program are fixed and there is no way to express signal deletion.
- **Signals graphs are static** – Any signal graph constructed in our program remain the same during a programs entire execution with no dynamic changes in behaviour.

These three properties are the some of the main areas in which we see differences in the semantics of FRP models. However, it's important to note that in the case of all of them no solution is necessarily better than another. Each of them comes with their own trade-offs

**2.2.3 Arrowized FRP**

The main model of FRP that I am interested in for the purposes of this project is **Arrowized FRP (AFRP)** [22]. AFRP is a FRP model formulated by Henrik Nilsson, Antony Courtney, and John Peterson at Yale in 2002. It makes extensive use of John Hughes’s Arrow combinators [12] to maintain the expressiveness of Classic FRP and fix some of its issues. However, unlike in Classic FRP, Signals
(Behaviours) and Events are not defined as first-class values. Instead, it defines signal functions as first-class values which allows for higher-order network descriptions.

AFRP declares a new type $SF$ to represent signal functions. This can be thought of as simply a function of a Signal to a Signal.

$$SF \ a \ b = Signal \ a \rightarrow Signal \ b$$

Similar to before, continuous time-variant values can be thought of as functions of time:

$$Signal \ a = Time \rightarrow a$$

The actual representation of the $SF$ is usually hidden in AFRP implementations so it is not possible to directly build our own signals or apply them to signal functions. Instead, a set of primitive signal functions and special composition operators are provided [10].

Again, it can be easiest to picture these making up a Signal Graph where Signals are represented as wires and Signal Functions as boxes which take a Signal as input and output another. It is important to note that this model and the types above are just conceptual models, their actual implementation is very different.

Arrows

In AFRP, Signal Functions are made an instance of the Arrow framework proposed by John Hughes in order to perform composition. Arrows can be described as a generalization of monads that allow for non-linear composition to be expressed in a “point-free” style. A simple example of this style would be if we wanted to combine two functions $f1 :: a \rightarrow b$ and $f2 :: b \rightarrow c$. Instead of writing:

$$g :: a \rightarrow c$$

$$g \ x = f2 \ (f1 \ x)$$

We can write it in point-free style using the function composition operator $(\cdot)$ as follows:

$$g = f2 \ . \ f1$$
We describe this code as “point-free” because the values (points) passed to and returned from a function are never directly manipulated [28]. We build functions by composing them with others.

To perform this at the level of Signals, we first need to define a primitive operator to lift functions to operate on them:

\[
\text{arr} : : (a \rightarrow b) \rightarrow \text{SF} a b
\]

This allows to take a function defined on primitive types and use it as a signal function. When then need a primitive combinator for composing signal functions:

\[
(\gg\gg) : : \text{SF} b c \rightarrow \text{SF} c d \rightarrow \text{SF} b d
\]

This allows us to write a version of the function \( g \) that operates on signal functions by defining it as follows:

\[
gSF : : \text{SF} a c
gSF = \text{arr} f1 \gg\gg \text{arr} f2
\]

These two operators are defined as part of John Hughes’ Arrow typeclass. Conceptually, if we think of a Monad \( m \) of parametrised type \( a \) as representing a computation delivering a result of type \( a \), then we can think of an Arrow \( a \) of parametrised type \( b \ c \) as representing a computation with input of type \( b \) delivering a result of type \( c \).

As we’ve seen already, different combinators are used to compose functions without needing to make direct reference to the functions’ values. Hughes defines the Arrow class including the \( \text{arr} \) and \( \gg\gg \) operators as follows:

\[
\text{class Arrow a where}
\begin{align*}
\text{arr} &: : (b \rightarrow c) \rightarrow a b c \\
\gg\gg &: : a b c \rightarrow a c d \rightarrow a b d \\
\text{first} &: : a b c \rightarrow a (b,d) (c,d)
\end{align*}
\]

In Haskell, the \textit{class} keyword allows us to define a protocol for types to implement called a typeclass, similar to interfaces in Java. When something implements a typeclass we say it is an “instance” of it. The \textit{arr} method in the Arrow typeclass is analogous with the \textit{return} operator defined in the Monad class. It simply lifts a function to become a “pure” Arrow computation.
As we saw before, the infix operator \( \ggg \) allows us to compose Arrows by connecting the output of the first parameter to the input of the second one. It creates a new Arrow with an input type equal to that of the first parameter and output equal the second's. This is analogous with the Monadic \( \ggg = \) operator (bind). For any Monad \( m \), functions of type \( a \to m b \) are potential arrows.

In addition to composing Arrows linearly, it is often desirable to compose them in parallel i.e. to allow “branching” and “merging” of inputs and outputs. So far we have only defined operations analogous with Monadic ones, but by implementing the \texttt{first} combinator in the Arrow \texttt{typeclass}, it allows us to express non-linear composition, which Monads are unable to express.

Given an Arrow \( f \) this functions creates a new arrow which we'll call \( g \), representing a computation from a pair of inputs to a pair of outputs. The first component passed as input to this new Arrow \( g \) will then be passed as input to the original Arrow \( f \), while the second component remains unchanged. The output of \( g \) delivers a tuple containing the output of \( g \) and the second part of the input to \( g \).
Using these primitives, we can define other combinators in terms of them. For example, we can define a function called `second` as follows:

```haskell
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
```

This behaves similar to the first operator but given an Arrow `f` applies it to the second element in a pair of inputs.

With these two combinators we can then define another which applies two computations to a both components in a pair of inputs. This allows us to express parallel computation.

```haskell
(***): Arrow a => a b c -> a d e -> a (b, d) (c, e)
f *** g = first f >>> second g
```

Here, given two Arrows `f` and `g`, we apply `f` to the first element in a pair of inputs and `g` to the second.

Finally, we can also define a combinator which builds a pair from the result of the a single input passed to two arrows. This allows us to express branching computation.

```haskell
(&&): Arrow a => a b c -> a b d -> a b (c, d)
f && g = arr (\b -> (b, b)) >>> (f *** g)
```
Here we create a pair by duplicating a single input and apply two Arrows $f$ and $g$ using the $(***)$ operator. This allows us branches using Arrows.
Functions as Arrows

Now that we’ve seen how Arrows are defined, it’s interesting to note that one of the simplest examples of them are functions. In Haskell functions are treated just like any other type so we can make them an instance of Arrow using the \((\to)\) type constructor. When we write a function with type \(a \to b\) another way of expressing this would be \((\to) a \; b\). Bearing this in mind we make them an instance of Arrows as show below:

\[
\begin{align*}
\text{instance } \text{Arrow} \ (\to) \ \text{where} \\
\text{arr } f &= f \\
(f \ggg g) &= g \cdot f \\
\text{first } f &= \lambda(x,y) \to (f \; x, \; y)
\end{align*}
\]

When implementing \text{arr} here, converting a function into an Arrow is trivial. Functions are already arrows because they represent something which takes an input and delivers an output. To implement the \((\ggg)\) operator we can just use regular function composition. To implement \text{first} given a function \(f\), we just return a function which takes a tuple as input and applies \(f\) to the first element. The implementation of the other operators remain the same as above.

Modelling Using Combinators in AFRP

When modelling reactivity in AFRP, it can be better to think in terms of a commonly-used set of combinators rather than a minimal set [10]. Most implementations provide some convenient library functions to facilitate programming using Arrows. For instance in the AFRP library, Yampa, the following functions are provided:
Arrowized Functional Reactive Programming in Swift using Futures

\[
\begin{align*}
\text{identity} & : SF a a \\
\text{constant} & : b \rightarrow SF a b \\
\text{time} & : SF a \text{Time}
\end{align*}
\]

The `identity` signal function behaves the same as identity function in Haskell but is defined on the level of signals. The `constant` can be used to generate constant signals functions. Finally, the `time` signal function yields the current time.

Modelling of data flow in AFRP usually involves the composition of functions, this time using Arrows, to produce a graph of computation for signals to flow through. However there are some key differences in it’s design compared with First Generation FRP. Firstly, signals graphs in AFRP are dynamic so we are able change their behaviour by triggering events in a manner similar to what we saw in Classic FRP. As events occur we can swap, remove or add functions to our signals graph. Secondly, signals in AFRP are not connected to the world. This means we have no way representing something like the position of the mouse using it, instead we define our own signals inside of our program. As I mentioned before, AFRP doesn’t usually allow us to specifically create signals so this is usually done using functions like ones defined above.

In many ways, AFRP is more concerned with providing a means of abstracting functions and structuring code rather than handling interactions with external entities. This is a big part of why it is so useful and the reason it can be applied in many different domains. The tools it provides are not limited to operating only on signals and signals functions, which is why we can make use of them in this project.

### 2.3 Futures & Promises

Modern applications are highly event-driven, which often results in asynchronous code that is difficult to read and maintain as the size of the program grows [20]. A common pattern when writing asynchronous code as we saw is to use deeply-nested callbacks where each callback has a dependency on data returned from a previous asynchronous invocation. Rather than an application blocking on a potentially non-deterministic event, the execution semantics of the application shifts from the programmer, relinquishing control of the event when the invocation happens and registering a method with the intent of reacting when the event is performed [18].
The problem with this, as we saw before, is that the control of our program becomes inverted: instead of the application controlling itself, interaction with external entities determines what computations happen and when. It removes our ability to think in terms of familiar sequential algorithms. Heavy use of callbacks in large-scale applications can lead to an overly complicated flow of execution and makes it difficult to write code that can be easily composed.

One attempt to solve this problem is through the use of Futures and Promises. The term *promise* was first proposed by Daniel P. Friedman and David Wise in 1976 [9] describing a construct used for synchronization in concurrent systems. A similar concept called a *future* was described Henry G. Baker, Jr. and Carl Hewitt in 1977 [1].

2.3.1 Futures

Futures provide a convenient way to reason about performing many asynchronous operations in an efficient and non-blocking way without using nested callbacks. Put simply, a Future is a proxy object representing a result that has yet to be computed. Futures encapsulate asynchronous actions, allowing us to return a value from a computation, despite not computing it yet. Since they were first formulated, futures have become an important feature in many languages, including some functional ones. The main implementation I focused on for the purposes of this of this project was the one included in the standard library for the functional programming language Scala [25].

In Scala, a Future has three possible states: pending, rejected or resolved, representing an computation that has not been completed yet, one that has failed or one that has succeeded, respectively. If the computation has not yet completed, we say that the Future is not completed. If the computation has completed with a value or with an error, we say that the Future is completed. An important property of Futures is that they may only be assigned once. Once a Future object is given a value or an error, it becomes immutable.

In Scala we denote a Future of type \( a \) as `Future[a]`. For example, if we have a value of type `Future[Int]`, this represents a value of type `Int` that has not been computed yet. The easiest way to create a Future to perform a computation asynchronously is using the `future` function:

```scala
val fibFuture: Future[Int] = future {
  fib(1000)
```

24
Here, we are creating a new value `fibFuture` of type `Future[Int]`. We pass a block of code to the `future` function to be performed asynchronously which returns a Future. This future is assigned to `fibFuture` and will eventually contain the result of the block of code we passed it.

Now that we've seen how to start an asynchronous computation to create a new future value, we need to be able to use the result once it becomes available, so that we can do something useful with it. Futures allow us to attach a block of code to handle it's completion. This completion handler is of type `Try[T] => U` where `Try[T]` is an abstract data type with two cases: `Success[T]` when it holds a value and otherwise `Failure[T]`.

```swift
fibFuture.onComplete {
    case Success(res) => println("Result: " + res)
    case Failure(err) => println("Error: " + err.getMessage)
}
```

In this example, we are calling the function `onComplete` using an infix notation to add a completion handler to `fibFuture` from the previous example. This allows us to handle the result of both failed and successful future computations. However, if we'd prefer to only handle the result of computations that succeed or ones that fail, we can use the `onSuccess` and `onFailure` functions:

```swift
fibFuture onSuccess {
    case res => println("Result: " + res)
}

fibFuture onFailure {
    case err => println("Error: " + err)
}
```

Futures also provide a set functions of “higher order functions” (a function which takes another function as input) which allow a for a number of operations to be performed on them. Higher-order functions allow us to encapsulate common patterns of design into functional forms expressed using types. One of the basic higher-order functions is map, which takes a future of type `A` and a mapping function of type `A => B` to produce a new future of type `B`. We use it to return a new future which is completed with the mapped value of the future being operated on. This can be thought of as function of Future to a Future.
For example, if we wanted to map the result of our fibonacci computation to a boolean based on whether the number is even or odd, we could write it as follows:

```swift
fibFuture.map { res =>
    res % 2 == 0
}
```

As we saw before, there might also be cases where we want to perform an asynchronous computation which depends on the result of another. This is what generally leads us to having a structure of deeply-nested callback inside our code. To eliminate the need to write this kind of structure, futures provide a method called `flatMap`. This lets us linearly chain a series of asynchronous computations, by applying a function of type \( A \Rightarrow \text{Future}[B] \) to a `Future[A]`.

For example:

```swift
val f = logIn(username, password) flatMap {
    user => getPosts(user)
} onSucess {
    posts => // do something with posts
} onFailure {
    err => // handle error
}
```

In this example we have a function called `logIn` which returns a Future representing a user. We then use `flatMap` to fetch the user’s posts after `logIn` has completed. This creates a new Future which will complete with the user's posts. We then define some completion handlers to do something the posts and handle errors. This function is analogous with the `(>>=)` operator in Haskell. This is because Futures are Monadic, which as we’ll see in the implementation chapter means the can be easily made an instance of Arrows.

### 2.3.2 Promises

So far we have only looked at Future objects created by asynchronous functions which return Futures. However, futures can also be created using Promises [25]. If we think of futures as a type of read-only placeholder object for pending results, promises can be thought of as writable containers around a future that can be used to complete it. Promises provide two methods, success (or
Arrowized Functional Reactive Programming in Swift using Futures

resolve) and failure (or reject) which can be used to complete a future. This future completed by a promise \( p \) is usually accessed using the future method: \( p\).future. This can be used to create a model for the producer-consumer pattern in a concurrent system. For example:

```swift
val p = Promise[Int]
val f = p.future

val producer = future {
    val r = produceSomething()
    p.success r
}
val consumer = future {
    f.onSuccess {
        case r => doSomethingWithResult(r)
    }
}
```

In this example, we create a promise, obtain its future and begin two asynchronous computations. The first performs some computation which delivers a value \( r \), and is used to complete the future \( f \), by fulfilling the promise \( p \). The second reads the result \( r \) from the completed future \( f \) and performs some computation using it. Promises can also be used to encapsulate an existing function that is not designed to work with futures into one that does. For example, if we have a function that does some work and takes a callback which we want to use with futures, we can encapsulate it using a promise:

```swift
def futureDoSomething(input: Int) => Future[Int] {
    val p = Promise[Int]
    doSomething(input, { result => p.success result })
    p.future
}
```

Here we are creating a promise called \( p \) to wrap the function doSomething which takes a callback. Inside the callback we give it, we are calling success on \( p \) to complete its future which is returned by the our function futureDoSomething.
2.4 Summary

In this chapter I introduced some advanced abstractions from mixed software design domains that are relevant to this project: Functional Reactive Programming, Arrows and Futures.

FRP provides us with a means of modelling values which change state during execution in a pure and declarative manner. By being orientated around data flow, FRP allows us to treat these values that change state as signals to handle propagation of change over time. It also allows us to lift regular functions to the level of signals so signals can be operated on and transformed in different ways.

Arrowized FRP makes use of the Arrow typeclass to provide powerful non-linear composition. By operating at the level of signal functions we can abstractly define computation using Arrows as something which takes an input and delivers an output to facilitate composition. This allows us to branch and merge signals, creating a graph of computation for our data to propagate through as it changes over time.

Futures provide us with a way of representing values that we don’t currently have but may come available in the future. They also provide a practical way of handling success and failure cases. This makes them useful when writing asynchronous code. Their counterparts, Promises, allow us to write to Futures and wrap existing functions that don’t use them.

Each of these technologies allows us to model real-world interactions in different ways, placing an emphasis on types and functional programming techniques. In the next section, I’ll be discussing how we can bring them together using Swift.
Chapter 3

Design

3.1 Overview

For the purposes of this project I was interested in taking the model used in FRP for dealing with time-variance and applying it to Futures instead of Signals. This gives us a better framework for modelling an asynchronous system. In this chapter I will be demonstrating the motivation for modelling asynchronous code in this way. I'll also be showing how this framework can be used to solve some problems and give examples of the style of code written using it.

Note: From this section onwards, we will primarily be focusing on Swift rather than Haskell or Scala. All code samples will be given in Swift [16].

3.2 Function Composition

The primary goal of the framework is to provide better composition of functions that interact with asynchronous resources. This is achieved in two different ways using Arrows. Firstly, we can compose functions defined on futures using the (>>>) operator. As we saw in the Background section, the (>>>) operator takes two Arrows and composes them together. For example, we might have two functions \( f \), which maps a Future of type \( A \) to a Future of type \( B \); and \( g \) which maps a Future of type \( B \) to a Future of type \( C \). Using the (>>>) operator we can compose them to make a new function which maps a Future of type a
to a Future of type \( c \).

In the following example, we have two functions: \( \text{isOdd} \) which maps a \( \text{Future}\langle\text{Int}\rangle \) to a \( \text{Future}\langle\text{Bool}\rangle \) based on whether the result is odd or even; and \( \text{toString} \) which maps this Bool to a printable String. We compose these functions to create new one which takes in a \( \text{Future}\langle\text{Int}\rangle \) and outputs a \( \text{Future}\langle\text{String}\rangle \) containing a String representation of whether the result was odd or even.

```swift
func isOdd(future: Future<Int>) -> Future<Bool> {
    future.map {$0 % 2 == 0}
}

func toString(future: Future<Bool>) -> Future<String> {
    future.map {
        if $0 == true {
            return "Even"
        } else {
            return "Odd"
        }
    }
}

let composed: Future<Int> -> Future<String> = isOdd >>> toString

let result: Future<String> = composed(someFuture)
```

This example illustrates the basic concepts of chaining maps but the real power of this approach comes form utilising other operators. In the following example I have introduced two new operators which we have not seen before: \( \text{zip} \) and \( |> \). The \( \text{zip} \) function simply takes two futures and returns a new Future which will contain a tuple with the values of both of them if they are completed successfully. We can represent its type as \( (\text{Future}\langle A \rangle, \text{Future}\langle B \rangle) \rightarrow \text{Future}\langle (A,B) \rangle \).

The \( |> \) operator is simply used to apply an input to a function. So, instead of writing \( f(x) \), we can write \( x |> f \), for cleaner syntax when working with arrows.

```swift
func f(future: Future<Int>) -> Future<Int> {
    future.map {$0 + 1}
}
```
func g(future: Future<Int>) -> Future<Int> {
    future.map { $0 - 1 }
}

func add(future: Future<Int, Int>) -> Future<Int> {
    future.map { $0.0 + $0.1 }
}

let result: Future<Int> = zip(future1, future2) |> (f *** g) >>> add

In this example, we are zipping two futures of type Int. We apply a function \( f \) to the value of the first one a function \( g \) to the value of the second one using the \((***)\) operator. The pair of results are then added together. This demonstrates the kind of non-linear composition that Arrows allow us to express declaratively. Non-linear composition is important in a reactive system because our programs often have to respond to multiple events which occur at different time periods. When different events occur we often want to trigger different procedures based on the event.

Another way to express this would be to use functions defined on non-Future values by lifting them to the level of Futures using a function called liftF:

func f(value: Int) -> Int {
    return value + 1
}

func g(value: Int) -> Int {
    return value - 1
}

func add(values: (Int, Int)) -> Int {
    return values.0 + values.1
}

let composed: (Int, Int) -> Int = (f *** g) >>> add

let result: Future<Int> = zip(future1, future2) |> liftF(composed)

This example delivers the exact same result as the previous one, but we're composing functions defined on non-Future values and lifting them using liftF.
before the compose function is applied. This is useful to us for three reasons. Firstly, it allows us to write functions that are entirely pure like the ones above i.e. given the same inputs they always deliver the same outputs and have no side-effects. This makes them extremely easily to test \cite{4}. In order to check that the above code behaves as expected we no longer would need to create futures to pass to each of the functions to test them. Instead, we can just pass them primitive values, which are what they actually operate on. As well as this, their results are not dependant on any global state in our program. Secondly, it makes it much easier to reuse code. By not defining the functions on Futures, we can reuse them in other areas of our code and other projects. In the example we saw in the introduction using nested callbacks, each callback was tightly coupled with the one coming before it. They can not operate without having the data of the previous callback in their scope. However in our example above, we have eliminated this problem by using composition. Each function can operate independant of the others/ This means that functions can be reused and, in the long run, means we don’t have to write as much code. Finally, if we define a new operator \((\sim>)\), which takes in a Future of type \(A\) and function of type \(A \rightarrow B\) which we lift before applying, we can re-write the last two lines of the previous example as just:

\[
\text{let result: Future\langle Int \rangle = zip(future1, future2) \sim> (f *** g) >>> add}
\]

The interesting thing about this is that if we wanted to use our composed function with non-Future values, we simply write it as:

\[
\text{let result: Int = (value1, value2) |> (f *** g) >>> add}
\]

The only things that have changed are our input and the operator we use to apply it. The functions we defined before operate on non-Future values so we can apply a primitive value as input to the composed function using the \((\mid>)\) operator. Effectively, our code has changed from operating on asynchronous values to operating on synchronous ones with almost no code changes required at all. This is incredibly powerful because it means that we can reason about and structure asynchronous code in the exact same way as we can with synchronous code. Our approach to modelling asynchronous problems can be the same as our approach modelling synchronous ones, which is what we’re most familiar with. In contrast, if we had written this code using callbacks it would look something like this:
Arrowized Functional Reactive Programming in Swift using Futures

```swift
getFirstValue { value1 in
  getSecondValue { value2 in
    let newValue1 = f(value1)
    let newValue2 = g(value2)

    return newValue1 + newValue2
  }
}
```

This requires much more code than before and there exists a tight coupling between the callback of the first function and the second one. The second callback requires \texttt{value1} to be included in its scope for it to be work. As well as this, we have no way of using this model for synchronous code.

### 3.3 Error Handling

Any time we interact with an external entity there is a possibility that the operation we are performing can fail. In order for our code to be safe, we must account for this possibility every time we interact with one by handling both the success and failure cases. When writing code modelled using callbacks this is typically done by passing two values to callbacks, first, the value returned from the external entity and second, an error, if one occurred, otherwise equal to nil. In Swift we represent both these values using Optionals. For example:

```swift
sendRequest { optionalData: Response?, optionalError: NSError? in
  if let err = optionalError {
    // handle failure
  } else {
    // handle success
  }
}
```

In this example, we send a request for some data and specify a callback which is passed an Optional value representing a response and another representing an error. We handle the success and failure cases separately by performing an optional binding on \texttt{optionalError} to test if one occurred. If we wanted to send another request after this, we would perform it inside the success case and would need to do the same kind of error checking again. This can create a lot of repetition and redundancy in our code as it increases in complexity and
Arrowized Functional Reactive Programming in Swift using Futures makes it more difficult to read. One of the other primary goals of this framework is to help solve this problem, which was achieved using Futures.

As we saw in the background section, Futures provide us with a convenient way to handle success and failure cases. If we were to rewrite the code from the last example using Futures in Swift, it would look something like this:

```swift
requestFuture.onSuccess {
    // handle success
}.onFailure {
    // handle failure
}
```

In Scala you are able to specify multiple completion handlers, however it does not make any guarantees about the order they will executed in. However, in this implementation, the order you specify them matches the order they are executed in. This is useful if we want to perform multiple actions upon completion of a future. For example:

```swift
requestFuture.onSuccess { res in
    handleSuccess1(res)
}.onSuccess { res in
    handleSuccess2(res)
}.onSuccess { res in
    handleSuccess3(res)
}
```

In this example, once the Future is completed, `handleSuccess1` will be called first, followed by `handleSuccess2` and finally `handleSuccess3`. The same would apply to failure handlers. This allows us to separate the code we use in completion handlers into a sequence of handlers.

In the background section we also saw that Futures are Monadic, so they can be chained together using the `flatMap` method, as shown here:

```swift
requestFuture1.flatMap {
    requestFuture2
}.flatMap {
    requestFuture3
}.onSuccess {
    // handle success
}.onFailure {
```
Each time use the `flatMap` we are getting a new Future, capturing the two being chained together, rather than changing the state of the current we are operating on. This emphasises the use of values rather than maintaining mutable state in our code. It also allows us to define one success and failure handler for the entire chain.

In cases where we want to handle success and failure at each stage in the chain separately, we can abstract each part to separate functions to make the code more readable. For example:

```swift
func sendRequest2(future: Future<Response>) -> Future<Response> {
    return future.flatMap {
        requestFuture2
        }.onSuccess {
            // handle success
        }.onFailure {
            // handle failure
        }
}

func sendRequest3(future: Future<Response>) -> Future<Response> {
    return future.flatMap {
        requestFuture3
        }.onSuccess {
            // handle success
        }.onFailure {
            // handle failure
        }
}

let future = requestFuture1 |> sendRequest2 >>> sendRequest3
```

As we see, this kind of monadic is easily captured by Arrows. The other advantage of expressing code like this using them is that it fits in with Swift more idiomatically. Swift provides a feature called `Optional Chaining` which allows us to unwrap different kinds of nested Optionals. This effectively allows us to express multiple optional queries in a single expression. Optional chaining is denoted using a question mark (?) after the optional value on which you wish to call a property, method or subscript on. For example:
if let roomCount = john.residence?.secondFloor?.numberOfRooms {
    println("John’s residence has \(roomCount) room(s) upstairs."")
} else {
    println("Unable to retrieve the number of rooms.")
}

In this example we are declaratively expressing a sequence of Optional queries. If any of them fail we go to the failure case otherwise continue to the success case. This is an example of programming paradigm used for error handling, often referred to as Railway Orientated Programming [31], which is exactly the same type of error handling we are expressing using Arrows with Futures.

### 3.4 Summary

In this chapter, we introduced the key design features of the framework presented in this project. These are function composition and error handling. We've now seen the benefits that this framework offers us when modelling asynchronous code and have a sense of how we can go about writing code using it.
Chapter 4

Implementation

In the previous section we weren’t introduced to the core design features of this project. In this section we’ll look at how we can implement them in Swift.

4.1 Arrows

If we look at the signatures for each of the Arrow operators in Haskell we see they are defined as follows:

\[
\begin{align*}
\text{arr} & : \text{Arrow} \Rightarrow (b \rightarrow c) \rightarrow a \rightarrow b \rightarrow c \\
(\ggg) & : \text{Arrow} \Rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow d \\
(\lll) & : \text{Arrow} \Rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow d \\
\text{first} & : \text{Arrow} \Rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow (b, d) \rightarrow (c, d) \\
\text{second} & : \text{Arrow} \Rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow (d, b) \rightarrow (d, c) \\
(\bigstar) & : \text{Arrow} \Rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow d \rightarrow e \rightarrow a \rightarrow (b, d) \rightarrow (c, e) \\
(\&\&\&) & : \text{Arrow} \Rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow (c, d)
\end{align*}
\]

These type signatures can be used to guide our implementation of them in Swift. Taking \(\text{arr}\) as an example, by reading its type signature we can see it is a function that takes another function of type \(b \rightarrow c\) and returns an Arrow encapsulating this function.

In order to provide the type of function composition we saw in the design section, we only need to provide Arrows at the level of functions, rather than at the level of both functions and Future functions.

In Haskell, the \(\text{arr}, (\ggg)\) and \(\text{first}\) operators are defined as part of the Arrow typeclass (analogous with a protocol in Swift), allowing the others to be
defined in terms of these three for all Arrow instances. Haskell allows us to make functions an instance of the Arrow typeclass using the (->) type, however this is not currently possible to implement in Swift due to how it handles types [21]. This means the Arrow operators we define in Swift will be globally available functions. From a programming perspective this makes little difference. From a semantic one though, it means values we use with Arrow operators will have all of the properties exhibited by Arrows but we will have no way of making them be of type Arrow. This makes little difference from a programming perspective because functions will still be type checked by Swift's compiler. For this reason we'll have no need for the arr operator, as we're just operating at the level of functions.

We can define custom operators using the operator keyword in Swift. Operators can only be defined with non-alphanumerical names, so first and second must be defined as normal functions. Swift also lets us define the associativity and precedence of our custom operators.

```swift
infix operator >>> { associativity right precedence 200 }
infix operator <<< { associativity left precedence 200 }
infix operator &&& { associativity right precedence 150 }
infix operator *** { associativity right precedence 150 }
```

Here we are specifying the composition operators with a higher precedence than the branching and parallel operators so we can compose a series of functions before branching them or using them in parallel.

For example, we can write the following:

```swift
test1 >>> test2 *** test3 >>> test4
```

Rather than:

```swift
(test1 >>> test2) *** (test3 >>> test4)
```

When using the (>>>) operator we expect to take a function of type A -> B, another of type B -> C and compose them to create one that is of type A -> C. Assuming we keep our function implementations pure, there's actually one thing we do here to satisfy the type constraints. Given the two functions f and g, our only option is to return a closure which applies them to the closure's input. This style of programming is often referred to as Type Driven Development [2]. By carefully deciding the types of functions in our code, we are able to
guide their implementation and in some cases restrict them so that the only
option available is the correct one. This type of development is only possible
in languages with expressive enough type systems. In this section we’ll see that
Swift’s type systems is strong enough to support this style of development.

Bearing this in mind, we can implement the (>>>) operator as follows.

```swift
func <<< <A, B, C>((f: A -> B, g: B -> C) -> (A -> C) {
    return { g(f($0)) }
}
```

The (<<<) operator behaves the same as the previous operator but composes
in the other direction so we can define it as follows:

```swift
func <<< <A, B, C>((f: B -> C, g: A -> B) -> (A -> C) {
    return g >>> f
}
```

The first function takes a function `f` of type `A -> C` as input and outputs
a closure of type `((A, B)) -> (C, B)`. The closure returned takes two inputs of
type `A` and `B` as input and applies `f` to the first one, mapping it from `A` to `C`.
In the Haskell implementation, a tuple is used for the closure’s input, however
in Swift multiple function parameters are implicitly passed using a tuple. This
allows us to treat the two elements inside the tuple as separate arguments, so
we can implement it as follows:

```swift
func first<A, B, C>((f: A -> C) -> ((A, B)) -> (C, B) {
    return { (f($0), $1) }
}
```

The second function can be defined in terms of `first`, by first declaring a
function called `swap` which takes a tuple containing two values as input and
swaps the two values. Using this function we can the define `second` as the
composition of `swap`, `first`, and `swap` again.

```swift
func swap<A, B>(pair: (A, B)) -> (B, A) {
    return (pair.1, pair.0)
}
```

```swift
func second<A, B, C>((f: B -> C) -> ((A, B)) -> (A, C) {
    return swap >>> first(f) >>> swap
}
```
With \texttt{first} and \texttt{second} defined, we can use them to implement the parallel operator (**\texttt{***}**). Given two functions \( f \) and \( g \), this operator returns a closure which takes a pair of inputs and applies \( f \) to the first element and \( g \) to the second.

\[
\text{\texttt{func *** <A, B, C, D>(f: A -> C, g: B -> D) -> ((A, B)) -> (C, D)}}
\]

\[
\{ \text{return first(f) >>\texttt{>>}} \text{second(g)} \}
\]

Finally to implement the branching operator \&\&\& we need to declare another function \texttt{duplicate} which takes a single value as input and returns a tuple with 2 elements, both equal to the input value. Using this we can implement \&\&\& as the composition of \texttt{duplicate} and the parallel application of it's two input function:

\[
\text{\texttt{func \&\& <A, B, C, D>(f: A -> B, g: A -> C) -> (A -> (B, C))}}
\]

\[
\{ \text{func duplicate(value: A) -> (A, A) \{ \text{return (value, value) \}} \}
\]

\[
\text{return duplicate >>\texttt{>>}} \text{(f *** g)}
\]

In this case, we were able to define \texttt{duplicate} as a nested function inside of \&\&\&. This is something we weren't able to do with the \texttt{swap} function because of a compiler bug. The \texttt{swap} function needed to be written using generic types which we declare after the function's name. However in Swift, writing nested generic functions causes the Swift compiler to crash, so we must declare it outside of the body of (**\texttt{***}**).

In the design chapter we specified two additional operators for function application, |\texttt{>}\rangle and \langle \rightarrow \rangle. These can be implemented as follows:

\[
\text{\texttt{func |\texttt{>} <A, B>(value: A, transform: A -> B) -> B \{ \text{return transform(value) \}} \}}
\]

\[
\text{\texttt{func \langle \rightarrow \rangle <A, B>(value: Future\texttt{<A>>, transform: A -> B) -> Future\texttt{<B>}}}
\]

\[
\text{\{ \text{return value |\texttt{>} liftF(transform) \}} \}}
\]
Arrowized Functional Reactive Programming in Swift using Futures

The function `liftF` will be defined in the section on Futures.

We have now fully implemented Arrows in Swift which can be used to provide composition of both primitive types and Futures.

4.2 Futures

As we saw in the background section, Futures have three states: pending, rejected and resolved. In Swift we can represent this by declaring a new enum called `Result`, which might look something like this:

```swift
enum Result<T> {
    case Success(T)
    case Failure(NSError)
}
```

Here we are representing the success case in the enum with an associated value of a generic type `T`. The failure case uses `NSError` for its associated value, which is an object included in Apple’s Foundation framework for representing errors.

However, unfortunately due to a bug in Swift’s compiler this code will not compile [23]. Swift needs know the full layout size of an enum at compile time, so we can’t use generics types for associated values as their size is unknown. In version 1.0 of Swift it was possible to get around this by making the associated value with the success case be of a function type `() -> T`. Swift treats functions as reference types so the size of a function’s value is equal to the size of a pointer in memory. Marking this function using the `@autoclosure` compiler directive would then allow us to a primitive value for the success case by which gets automatically converting to a closure. However, in Swift 1.2 the semantics of `@autoclosure` have changed so we must use a class to wrap the value for the success case. Similar to functions, Classes are reference types in Swift so the size of a class in memory is always equal to the size of a pointer. Using the a wrapper, the `Result` enum now looks like this:

```swift
class Box<T> {
    let value: T
    init(value: T) { self.value = value }
}
```
The disadvantage of this is that when creating an instance of `Result` with the success case we must write the following:

```swift
let success = .Success(Box(value))
```

But by defining an initializer inside the `Result` enum, we can partially solve this problem:

```swift
eenum Result<T> {  
  case Success(Box<T>)  
  case Failure(NSError)  
  
  init(value: T) {  
    self = .Success(Box(value))  
  }  
}  
let success = .Result(value)
```

We can also define some utility methods for checking the state of a `Result` and accessing its associated values:

```swift
var isSuccess: Bool {  
  switch self {  
  case .Success(_:):  
    return true  
  case .Failure(_:):  
    return false  
  }  
}  

var isFailure: Bool {  
  return !self.isSuccess  
}

var value: T? {  
  switch self {  
  case .Success(value):  
    return value  
  case .Failure(_:):  
    return nil  
  }  
}  
```
Swift gives us access to two data types for structuring groups of data: Classes and Structs. Classes are used to define a primary actor in an object model, representing both its attributes and its interactions. In the case of stateful objects like Futures, this makes them an appropriate choice. We can define the beginnings of a Future class as follows:

```swift
class Future<T> {
    var result: Result<T>?

    var isCompleted: Bool {
        return self.result != nil
    }

    var isPending: Bool {
        return self.result == nil
    }

    var isSuccess: Bool {
        return self.result?.value != nil
    }

    var isFailure: Bool {
        return self.result?.error != nil
    }
}
```
Here we are representing the pending state of a Future by making the result an Optional type i.e. if a Future doesn’t have a result then it is considered pending. We also define some utility methods for checking the state of a Future.

Now, we must add support for specifying completion handlers. To do this we first define a property for storing completion handlers as an array of functions of type `Result<T> -> ()`. We can then define `onComplete`, `onSuccess` and `onFailure` as methods which append to this list:

```swift
var completionHandlers: [Result<T> -> ()]

func onComplete(handler: Result<T> -> Void) -> Future<T> {
    if let result = self.result {
        callback(result)
    } else {
        self.completionHandlers.append(handler)
    }

    return self
}

func onSuccess(handler: T -> Void) -> Future<T> {
    return self.onComplete {
        if let value = $0.value {
            handler(value)
        }
    }
}

func onFailure(handler: NSError -> Void) -> Future<T> {
    return self.onComplete {
        if let error = $0.error {
            handler(error)
        }
    }
}
```

If a Future has already completed when one of these methods is called, the handler will be immediately invoked. In the above code, we define `onSuccess` and `onFailure` in terms of `onComplete` but only allow them to execute the handler if the Future is in the correct state. This allows us to keep all handlers stored inside the same array, so we guarantee the order they are specified in.
matches the order they are executed. Each of these functions return `self` so operations can be chained together.

With the completion handler being stored correctly, we can now add support for invoking them. To do this we simply define a method called `complete` which takes a `Result` as input and transitions the Future to the completed state:

```swift
func complete(result: Result<T>) {
    if self.isCompleted {
        return
    }
    self.result = result
    for callback in self.completionHandlers {
        callback(self.result!)
    }
    self.completionHandlers.removeAll(keepCapacity: false)
}
```

When the Future transitions to the completed state we invoke each of the completion handlers using the result and remove them from the array.

We now just need to add support for mapping and `flatMap`ing Futures, which we can do using the `complete` function we just defined. However, first we need add a map method to the `Result` enum, which can be defined as follows.

```swift
func map<U>(f: T -> U) -> Result<U> {
    switch self {
    case .Success(let box):
        return .Success(f(box.value))
    case .Failure(let error):
        return .Failure(error)
    }
}
```

Swift lets us match against and unwrap associated enum values using a `switch` statement as shown here. Given a function `f`, when a `Result` is equal to `Success` we apply `f` to the associated value.

Using the map method above we can define a similar function for Futures:

```swift
func map<U>(transform: T -> U) -> Future<U> {
    let future = Future<U>()
    return future.flatMap{ f in
        return map<U>({ $0 in
            return f($0)
        })
    }
}
```
Arrowized Functional Reactive Programming in Swift using Futures

```
self.onComplete { value in
    future.complete(value.flatMap(transform))
}

return future
```

Here, we are returning a new future which is completed using the mapped result of the future the method is being invoked on. We can then define the `flatMap` method as follows:

```swift
func flatMap<U>(transform: T -> Future<U>) -> Future<U> {
    let future = Future<U>()

    self.onComplete() { value in
        let newFuture: Future<U>

        switch value {
        case .Success(let box):
            newFuture = transform(box.value)
        case .Failure(let error):
            newFuture = Future<U>()
            newFuture.complete(.Failure(error))
        }

        newFuture.onComplete() { value in
            future.complete(value)
        }
    }

    return future
}
```

This method behaves similarly conceptually to the `map` function we defined before it. We can think it as a function mapping a `Future<T>` to a `Future<Future<U>>` and then “flattening” it into a `Future<U>`. The flattening is done by creating a separate future which is completed using the value of the inner one. As we saw in the previous chapter, this allows us to chain futures together.

With these functions defined we now have a fully operational implementation of Futures. However we still need to define two functions, `liftF` and `zip`
from the design section which we can write as follows:

```swift
func liftF<A, B>(f: A -> B) -> Future<A> -> Future<B> {
    return {
        $0.map { res in f(res) }
    }
}

func zip<A, B>(f1: Future<A>, f2: Future<B>) -> Future<(A, B)>
{
    let future = Future<(A, B)>()

    f1.onComplete { result1 in
        switch result1 {
        case .Failure(let err):
            future.complete(.Failure(err))
        case .Success(let boxedValue1):
            f2.onComplete { result2 in
                switch result2 {
                case .Failure(let err):
                    future.complete(.Failure(err))
                case .Success(let boxedValue2):
                    future.complete(Result((boxedValue1.value, boxedValue2.value)))
                }
            }
        }
    }

    return future
}
```

The first function `liftF` lifts a function to one defined on Futures by using it in a map. The second one, `zipF`, given two futures, returns a new future that will complete with both their values once they have completed.

### 4.3 Promises

Typically the `complete` method we defined in our Future class is not made available to other classes outside the framework to ensure using access-modifiers. This ensures that Futures are read-only. However Promises can act as a proxy for writing results to a Futures and as such are given access to it. This allows us to define them as a small class wrapping a Future with methods for completing it:
4.4 Example Program

In the introduction to this paper I demonstrated how code quickly becomes cluttered when using nested callbacks. The example I gave was a theoretical program that opens a TCP connection to a host, sends a message and reads a response. Using the framework I have outlined in this chapter, here is how we might model this now:

```swift
let future = DeferredTCP.connect(host) |> writeSocket
         >>> readResponse
         >>> parseResponse

future.onSuccess { parsed in
    println(parsed)
}.onFailure { err
    handleErr(err)
}
```

Instead of using nested callbacks we are composing a series of function to be applied to a Future. This example shows us how little code we need to write and how much easier it is read when using this framework. This example was
mostly used for the purposes of demonstration, but in the following section we’ll be looking at some real-world code that uses this framework.

### 4.4.1 API Interaction

The code below is from a program I wrote for test purposes which fetches the number of repositories on GitHub written in Swift with over 1000 stars. It works by first sending a request to GitHub asking for all repositories written using Swift sorted by number of stars. This is uses a tiny class called `DeferredURLRequest` which simply creates a Future for a HTTP request by wrapping an existing standard library function using a Promise. HTTP Requests in Swift return an object of type `NSData` which simply represents unparsed data. The code includes three functions which are composed to operate on the data returned by the request. The first converts the data into a JSON object that can be more easily interacted with. The second filters all repos from the JSON with less than the value of `MIN_STARS` which we defined at the top of the program. It makes uses of “currying” so the function can be partially applied [14]. Finally, the last one returns the count of the now filtered repos. As we’ve come expect, each of these functions are entirely pure and decoupled from each other so they can be tested individually. The composition of these functions are then lifted to level of futures, and we handle the success and failure cases by specifying handlers.

```swift
typealias Repository = JSON

let MIN_STARS = 1000

func parseJSON(a: NSData) -> JSON {
    return JSON(a)
}

func filterRepos(count: Int)(json: JSON) -> [Repository] {
    return json["items"].array!.filter {
        let star_count = $0["stargazers_count"].integer!
        return star_count > count
    }
}

func countRepos(repos: [Repository]) -> Int {
    return repos.count
}
```
let requestURL = NSURL(string: "https://api.github.com/search/repositories?q=language:swift&sort=stars&order=desc")

let future = DeferredURLRequest.requestWithURL(requestURL!) ~> (parseJSON >>> filterRepos(MIN_STARS) >>> countRepos)

future.onSuccess {
    println("Number of Swift repos with 1000+ stars: " + String($0))
}
.onFailure {
    println($0)
}

There's a lot happening in this program, but using the expressive syntax provided by this framework we are able to express it extremely succinctly. This demonstrates how the framework presented in this project might be used in a real world scenario.
Chapter 5

Results

5.1 Conclusions

In the introduction to this report I posed the question: Is it possible to write functional programs in Swift and if so how does this benefit us as programmers? This project demonstrate that the answer to the first part of this questions is yes. Almost all of the code for it was written using pure, side-effect free functions and stays true to the three properties of well-designed functional programs I identified in the introduction.

The intention of the project was to create an Arrowized FRP framework for modelling asynchronous programming using Futures. In the design chapter we saw some of the benefits this framework offers us. Firstly it provides a powerful means of composing functions in a declarative manner. Functions can be composed at the level of both Futures and primitive types, with support for branching and performing computations in parallel. This means we are able to write pure functions defined on primitive types which we can compose and then lift to operate on Futures. The advantage of this is that it means we can model asynchronous code and synchronous code in the same declarative manner with little changes to our code, making it easier to reason about. It also means we can more easily test the functions that we write to transform Futures because they are just regular pure functions.

Furthermore, the model I have presented reduces the redundancy and repetition in asynchronous code. Previously, when using nested callbacks, our code
would continue to drift further and further right as we chained asynchronous operations. We also needed to check for errors at each stage, despite often handling them in the same way each time. Futures combined with Arrows provide us with a convenient way to chain asynchronous operations together and handle errors. Instead of repeatedly using the same error handler at each stage in a chain of operations, we can define success and failure handlers for the entire chain. In cases where we want to define different success and failure handlers at each stage, Arrows provide us with way of abstracting this in idiomatic way.

In conclusion, I believe that this project was a great success. With it, I have managed to create a model for asynchronous programming and reactive systems in Swift which I believe to be far better than the original approach of using nested callbacks. In the process I have also been able to demonstrate the power of some of the functional properties of Swift.

I believe the advantages of this project offer a glimpse into how functional programming can benefit programmers using Swift. This is just one example of the many possible functional paradigms that could be ported to Swift offering similar benefits. The exciting thing about Swift is that it is still so early in development that idioms have not been fully established yet. This means we as a community have the opportunity to develop them ourselves through experimentation. It’s entirely possible at this stage to start a project like this and have the designs it presents become commonplace, which is what makes this project interesting.

### 5.2 Limitations

The main limitations of this project are brought about by Swift’s compiler and how it handle generic types. Unfortunately because Swift is closed source and still so young, there’s not a lot of research or information on its compiler. However, based on my interaction with it, I identified that, during compilation, it initially creates placeholder types for all generic types used in the program it is compiling. It then performs a single-pass attempt to synthesize types for each of the placeholders. If any of them cannot be synthesized it will throw an error in the form “Could not infer type of $Tn” where n is a number assigned to the placeholder.

These types of errors can occur a lot when developing using this frame-
work and were a serious hindrance during its implementation. This can make programs very difficult to debug and often the only solution is to break down expressions into smaller ones with explicit type annotations to determine which one the compiler cannot infer the type for.

As we saw in the implementation section, the compiler also contains fatal bugs involving generics inside of enums and nested function. Both these that are problems that will require serious changes to Swift’s compiler to be fixed.

Swift also does not provide a way of using generic typed arguments inside closures. For example if we wanted to use an identity function in an Arrow composition instead of writing it as simply \( \text{\{ 0 \} } \) we must declare it as a function.

The other main limitation of this project is that it can be awkward to write code using Arrows. This is something that also effects Haskell implementations of AFRP. As the complexity of the expressions we write using Arrows increases, they can become cluttered, particularly when using operators other \((>>>)\) for forward composition.

5.3 Further Work

This project identified to me a number of potential areas for future work following it.

The first potential area to be explored would be in developing a full port of Arrowized-FRP to Swift. A large part of the reason I chose to implement an AFRP framework for working with Futures was that I was unsure if it would be possible to implement it in a fully expressive manner using Swift. However, this project has demonstrated that Swift’s type system is expressive enough to represent these kind of abstract functional paradigms. Upon realising this I quickly put together a small, test implementation of AFRP based off of Yampa. With more time put into developing the framework it could easily become a full-featured AFRP implementation. AFRP has already proven its usefulness in a variety of domains such as animation, sound synthesis and robotics [10], all of which could be potentially useful to Swift developers.

A second area worth exploring would be the development of nicer syntax for dealing with Monads and Arrows. Programs structured using Arrows can often suffer from being difficult to read. Similarly, when working with Futures, there can often be a lot of repetition in our code when we chain operations together.
using the `flatMap` method. The solution to both these problems is to develop nicer syntax for dealing with them. In Haskell the solution to this is called *Do Notation* and there exists a similar concept in Scala called *For Expressions.*

Rewriting the TCP example from the introduction in Swift with a hypothetical syntax like these might look something like this:

```swift
func connectTCPAndSend(host: Host, msg: String) -> Future<Response> {
    return for {
        socket <- DeferredTCP.connectHost(host)
        socket.sendMessage(msg)
        response <- socket.readResponse()
        yield response
    }
}
```

This code is analogous with the following which as we can see is not quite as clear:

```swift
func connectTCPAndSend(host: Host, msg: String) -> Future<Response> {
    let future = DeferredTCP.connectHost(host).flatMap { socket in
        socket.sendMessage(msg)
        .flatMap { socket in
            socket.readResponse()
        }
    }

    return future
}
```

Haskell provides similar syntax for working with Arrows called *Arrow Notation* as compiler extension that can be turned on [24]. Here’s a simple example of how it might look in Swift:

```swift
func addF(f: Future<Int> -> Future<Int>,
          g: Future<Int> -> Future<Int>) -> Future<Int> {
    return forA (x in
                 y <- f -< x
                 z <- g -< x
                 yieldA y + z
    )
}
```

The above code would be potentially analogous with the following expression:
f && g >>> liftF { (y, z) in y + z }

Although the previous sample involves more code, it’s easier to see what it’s doing. In the case of both of these syntaxes, it would not be possible implement them natively without updating Swift’s compiler extensively. However a potential solution to this would be to write a preprocessor supporting them, which could be packaged as a plugin for Xcode.

Finally, another area worth exploring would be the use Arrows in other Swift application domains. Arrows are an important part of AFRP but there use goes far beyond that. In his paper introducing them, John Hughes presents a number different use cases for them such as writing parsers and simulating logic circuits [12]. In this project I demonstrated how they can be used to compose functions defined on primitive types and lifted to level of other types, such as Futures. Haskell programmers make heavy use of them for these reasons, the offer us a huge amount compositional flexibility – more so than monads [13]. However, as we have seen this also applies to Swift, so I believe there is plenty of opportunity to apply them in other domains and explore the benefits they offer us.

These are just some of the areas that can be explored following on from this project.

5.4 Closing Remarks

I hope that my contributions and questions will help push the Swift community towards embracing more of it’s functional features. From a programming perspective, this projects presents the opportunity for reactive systems to be modelled more efficiently. From a research perspective, it demonstrates the potential Swift has for future work in experimenting with new paradigms and concepts. I look forward to seeing what the community has in store.
References


[26] SwiftyJSON. Swiftyjson – the better way to deal with json data in swift. https://github.com/SwiftyJSON/SwiftyJSON.


