Functional and Reactive Patterns in Idiomatically Imperative Programming Languages

JESPER SANDSTRÖM
Functional and Reactive Patterns in Idiomatically Imperative Programming Languages

JESPER SANDSTRÖM

Master in Computer Science
Date: June 8, 2018
Email: jesands@kth.se
Supervisor: Musard Balliu
Examiner: Mads Dam
Swedish title: Funktionella och reaktiva programmeringsmönster i imperativa programspråk
School of Electrical Engineering and Computer Science
Abstract

Functional and reactive programming patterns provide powerful sets of tools for dealing with complexity and scalability. These stand in stark contrast to the imperative programming patterns which permeate the industry. This report seeks to investigate the extent to which a set of common imperative programming languages can be used to implement such functional and reactive patterns, and what the implications of doing so are.

This is done by implementing and using a framework based on such patterns in Java, Kotlin, Objective-C, and Swift. The results show that this is possible in all of these languages, but the extent to which this can be considered idiomatic is questionable. Upholding immutability and referential transparency is highlighted as the main source of concern in this regard.
Sammanfattning

Funktionella och reaktiva programmeringsmönster förser utvecklare med kraftfulla abstraktioner för att hantera komplexitet och skalbarhet. Dagens industri förlitar sig dock till en majoritet på imperativa programspråk, där dessa mönster inte nödvändigtvis kan utnyttjas. Syftet med denna rapport är därför att undersöka hur sådana mönster kan tillämpas i imperativa programspråk.

Chapter 1

Introduction

Software construction is a complicated endeavour and the costs of mistakes is only increasing as our reliance on software increases. Faults in software can have significant economic impacts and potentially pose health hazards. Complexity in software arises not only from the problem domains, but also as a result of the programming strategies employed to model these domains. This report draws upon patterns and principles from functional programming to tackle this additional complexity.

Functional programming offers a range of benefits in software development, such as making programs easier to reason about [30], isolating and modelling effects [31], and improving testability. Reactive programming offers a means of modelling how a system should respond to events, such as those generated by user interaction. However, both paradigms impose a set of constraints which are not necessarily idiomatic to comply with or statically enforceable in imperative programming languages. For example, functional programming requires functions to be pure and data to be immutable, whereas reactive programming requires a means of defining a current system’s state as a function of events and previous states.

This thesis investigates the extent to which certain features of imperative programming languages can enable or hinder programming in a functional and reactive style. This is done by studying implementations and usages of a subset of Spotify’s Mobius framework [27] in four non-functional programming languages: Java, Objective-C, Swift and Kotlin. Mobius is a state-management framework which builds upon concepts from functional reactive programming and other con-
temporary tools such as Redux [2] and Elm [9].

1.1 Scope

This investigation is composed of three stages. The first involves defining Mobius in a high-level, purely functional language. This implementation serves as a reference point for the other languages. The second stage is to implement Mobius based on these semantics in Java, Kotlin, Objective-C and Swift. The third stage analyzes how the framework can be used in these languages.

The goal is to find out which concepts these languages are capable of expressing, and the degree to which this is practical. From a broader perspective, the goal is to identify the features which make languages appealing for programming in this style. Achieving this goal will involve studying the relative sizes of the solutions produced, the degree to which the solutions can be statically typed, limitations of the concurrency models in these languages and the possibility of addressing the greatest limitations with the use of meta-programming.

This project does not discuss the performance or memory usage of these implementations. It also does investigate the efficacy of this style of programming.

The official version of the Mobius framework is written in Java. This report will formalize and implement a framework based on a subset of the official framework. Inspiration is drawn from the official implementation where appropriate, but all design elements were reconsidered from the design up. This deviation from the official framework is done to focus on the core principles of these patterns. Henceforth, Mobius will refer to this report’s definition of the framework unless otherwise stated.

1.2 Motivation

This report’s case for adopting Mobius, and related patterns, in imperative languages is based on the following principals:

- **Referential Transparency** - Referentially transparent functions cannot depend on anything other than their arguments by definition. This simplifies the processes of reasoning about the composition of such functions, thereby enabling programming at a
higher level of abstraction. As an added benefit, this enables them to be tested in isolation which leads to fundamentally simpler tests which can be deterministically executed in parallel.

- **Declarativity** - A declarative approach enables the programmer to distinguish between what a program should do and how it should do it. For example, side-effects in Mobius are just represented as data. They do not carry out an action; they represent an intent to carry out an action. The actual evaluation of these effects is deferred to a specific part of the framework. The core business logic can therefore be verified in isolation. It also paves the way for optimizations such as batching or caching without changing the business logic.

Establishing the distinction between the representation of an effect and actually carrying out an effect enables equational reasoning. Equational reasoning would not be preserved if effects were simply carried out instead [31].

- **Immutability** - The state of immutable objects cannot be changed. They can therefore safely be shared across threads, and can make programs easier to understand [21]. This property is a cornerstone of functional programming and plays a key role in the Mobius framework.

### 1.3 Research Question

It is important to note that the principles outlined in section 1.2 can only be encouraged and not enforced in the languages used in this project. The type systems of these languages are not powerful enough to enforce referential transparency and even seemingly immutable objects can often be mutated through the use of reflection. Any violation of these properties can permeate through the entire system and thus invalidate any benefits [22]. These arguments are therefore presented as ideals rather than laws that invariably hold. The goal is to make these principles idiomatic. As such, the research question is:

How can certain language features of imperative languages enable or hinder programming in a functional, reactive and declarative style?
1.4 Hypothesis

The hypothesis is that the newer programming languages, Swift and Kotlin, will yield overall more concise solutions for both implementation and usage compared to Java and Objective-C. This prediction is based on the fact that these languages are newer, and they have as such benefited from decades of advancements in programming language design, and on their support for features such as anonymous functions. In the case of Kotlin, there is a focus on immutable by default data structures, support for pattern matching and terse syntax for declaring algebraic data types. For these reasons, Kotlin will likely perform best. This hypothesis would be invalidated if the results do not reinforce this assumption.

1.5 Contributions

This report investigates ways of programming in a functional and reactive style. Its main contributions are that it provides formalization of the Mobius framework in a purely functional language, a study of the language features which are most relevant to this style of programming, and a range of strategies for dealing with a lack of these language features.

The results show that the main point of contention between these languages lies in their ability to express algebraic data types. Various methods for compensating for the lack of algebraic data types are demonstrated, most notably by outlining a correspondence between Objective-C’s protocol and message passing systems to algebraic data types. The differences in the extent to which the implementations are statically typed are also noted. The outlier in this regard is, again, the dynamically typed approach taken in Objective-C. Overall, the differences between these languages can be minimized through the presented strategies.

1.6 Ethics and Sustainability

This thesis pertains to the fields of software engineering and programming languages. One goal is therefore to investigate ways of improving the how software is built by challenging the imperative idioms of
these languages. Creating software which is more maintainable and scalable is also more economically sustainable.

Another goal is to find ways to write programs that are easier to reason about. This could have indirect ethical implications in the sense that code which is easier to reason about may be more likely to be correct and secure, and as such may help address personal integrity and privacy issues.

1.7 Intended Audience

The recommend theoretical results of this thesis are recommended mainly to those who are interested in the fields of the programming languages, programming paradigms, and software architecture. This is because it outlines an architecture based on principles of functional and reactive programming, and provides tangible interpretations of these patterns in a range of imperative programming languages. The conclusions range from specific strategies in these languages to general reflections on imperative programming idioms. From a broader perspective, the general conclusions are recommended to any software engineer, since these conclusions deal with ways of writing software which is simpler to test and reason about. These motivations are outlined in detail in section 1.2.

1.8 Structure

The rest of the report consists of 6 chapters. The first of these chapters provides the theoretical context for the rest of the report (chapter 2). This is followed by a description of the methods used in this investigation (chapter 3), the set of results achieved (chapter 4), a discussion of the implications of the results (chapter 5), an investigation of related works (chapter 6), and finally a conclusion which provides an overall summary (chapter 7).
Chapter 2

Background

This chapter provides the theoretical background for this report. The general concepts are outlined first. The report then discusses these concepts within the context of this investigation. The rest of this thesis contains programming examples written in Haskell, Java, Objective-C, Swift, and Kotlin. The specifics of the syntax in these examples is less important, since the accompanying texts also convey the same meaning.

In this report, functional programming and imperative programming are compared. In the interest of avoiding ambiguity, these concepts will first be defined.

2.1 Functional Programming

Functional programming entails creating programs entirely through the composition of mathematical-like functions [19]. This approach forbids any side-effecting constructs such as global state, assignment statements or exceptions [32]. While seemingly limiting, this approach enables programmers to reason about their programs both formally and informally in accordance with mathematical laws. Technically speaking, these functions are only mathematical-like in the sense that they often only define a partial mapping from their domains to codomains. However, it has been shown that this technicality does not invalidate such reasoning [12].

Functional languages like Haskell are called pure since they only permit pure functions, whereas languages like Scheme are impure since they contain a number of side-effecting constructs [32]. The common-
ality between these languages is that they rely on the composition of functions as their primary tool for constructing abstractions. In the context of this report, Haskell is the main point of reference from this paradigm. Therefore, for the sake of brevity, functional programming languages is used interchangeably with pure functional programming languages.

2.1.1 Referential Transparency

A function is referentially transparent if it performs no side-effects and it always returns the same output for a given input [17]. One useful way of viewing such functions is as it being replaceable with an associative array (an array where each value has a key associated with it) where the keys are the input to the function and the values are the output.

2.1.2 Equational Reasoning

Referential transparency enables equational reasoning, i.e. "substituting equals for equals" [31] in all contexts. For example, a function can always be replaced by its implementation. This property of purely functional programming languages enhances both the programmer’s ability to reason about their programs [30] and enables the compiler to further optimize programs [24].

2.2 Imperative Programming

The defining characteristic of the imperative programming paradigm is the idea of structuring programs as a series of imperatives or commands. Calculations are performed through the results of performing this series of commands on a shared global state [22]. In this report, imperative programming languages refer to languages which make this style of programming idiomatic. I.e., languages which do not enforce immutability or referential transparency and make it possible to share globally mutable state across all parts of a program.
2.3 Programming Language Features

This section contains an overview of the programming language features which will be discussed throughout this report.

2.3.1 Algebraic Data Types

Algebraic Data Types (ADTs) are formed through combinations of two simpler types: sum types and product types. A sum type is the disjunction of two types: e.g., if a value can be either a string or an integer, then its type is the sum type created by strings and integers. Only one of these can be present at a time. Conversely, a product type is the conjunction of two types: e.g., a value may contain an integer and a string, in which case its type is the product of strings and integers.

An ADT is represented by the sum of several subtypes, which may in turn be sum-types or product types themselves. This can be better understood through the use of an example. Consider the type of an arithmetic expression where the expression is either a value or a binary operation on expressions. This can be represented with the following ADT in Haskell:

```haskell
data Expr a
    = Value a
    | Add (Expr a) (Expr a)
    | Subtract (Expr a) (Expr a)
    | Multiply (Expr a) (Expr a)
    | Divide (Expr a) (Expr a)
```

In this case, the expression is parameterized by the type a, so that the user can define which type to define expressions for. The five cases, `Value`, `Add`, `Sub`, `Multiply`, and `Divide` represent all the types that such an expression can have. These cases are called variants of the algebraic data type. ADTs are useful because they provide a means of encapsulating a set of choices within one single type.

2.3.2 Exhaustive Variant Checking and Pattern Matching

Recall that ADTs contain a set of variants. On their own, these provide a data-representation of an arbitrary structure. Variant checking
and pattern matching are complementary constructs to ADTs. They allow the programmer to define ways of interpreting the meaning of the ADT’s structure. Consider a function, `eval`, which evaluates an expression of integers:

```haskell
eval :: Expr Integer -> Integer
eval expr =
    case expr of
        Value n -> n
        Add expr1 expr2 -> eval expr1 + eval expr2
        Subtract expr1 expr2 -> eval expr1 - eval expr2
        [...] other cases omitted for brevity
```

By calling itself recursively, the `eval` function is able to traverse the entire structure of the ADT and evaluate it into a single value. This check is called exhaustive if the compiler requires the user to define what should happen for each variant.

The main takeaway here is that by using these constructs the representation of a program can be separated from its interpretation - the ADTs describe the structure of the desired computation, whereas the `eval` function defines one way of interpreting this ADT.

### 2.3.3 Anonymous and First-Class Functions

As the name suggests, anonymous functions are functions which can be declared without a name. For example, an anonymous function which adds two values can be expressed as follows in Haskell:

```haskell
\a b -> a + b
```

A language is said to have functions as first-class citizen if the language supports passing functions as arguments to other functions, returning functions from functions and the ability to store functions in data structures or refer to them with variables [1].

### 2.3.4 Immutable/Persistent Data Structures

Immutable data structures are data structures which cannot be modified after initialization. Any function which changes such a structure must return a new data structure instead of modifying the contents of the old one. However, this does not necessarily imply that the entire
data structure needs to be entirely copied. If the underlying data is guaranteed to be immutable, both old and new versions of the data structure can share their common parts. Data structures which utilize this property are called persistent.

2.3.5 Generics and Polymorphism

Polymorphism refers to a function’s ability to operate on a range of different types. Though different types of polymorphism exist, this report will specifically deal with parametric polymorphism. Parametric polymorphism refers to the ability to define functions which are defined uniformly without reference to specific types. This is useful when the mechanics of such a function are independent of the types it operates on. This enables the creation of type-safe and extensible abstractions. The imperative languages discussed in this report support parametric polymorphism to various extents through the use of a language feature typically called generics. Generics refer to functions and classes which are parameterized by types.

2.4 Programming Languages

Java, Kotlin, Objective-C and Swift were primarily chosen for this investigation because of their major role in mobile application development. However, their relevance extends beyond this field. They are all among the 20 most popular programming languages according to the PYPL index, with all (with the exception of Kotlin) appearing in the top 20 of the TIOBE index of programming languages as well. They are also representative of a progression of eras within programming language design, with Objective-C having been released in 1984, Java in 1995, Kotlin in 2011, and Swift in 2014. These factors combined make the languages good representatives of the types of imperative, object-oriented languages used in industry today.

2.4.1 Java

Java, initially developed by Sun Microsystems, compiles into bytecode for the Java Virtual Machine (JVM). The initial implementation lacked many features that would make it suitable for programming in a functional style, but many of these have been added in later releases,
notably the introduction of generics and anonymous functions. However, the use of immutable objects, algebraic data types, and pattern matching are still not idiomatic.

Gosling et al. [18] provide a full language reference. Java SE 8 was used for this report because the later versions are not compatible with the Android operating system. However, the results account for the features introduced in subsequent versions (9 and 10), and this decision did not prove to be significant.

2.4.2 Kotlin

Kotlin [20] is a programming language originally created for the JVM by JetBrains. It is fully interoperable with Java. There are, however, a list of notable differences from Java which are relevant for this project:

- Relatively terse syntax for creating algebraic datatypes in the form of sealed classes.
- Exhaustive case checking
- Pattern Matching in declarations
- Data classes with support for making immutable changes which return copies.
- Immutable maps, sets, and lists can be found in the standard library.
- Support for anonymous functions in all language versions.

Jetbrains [20] provide a full language reference. This report uses version 1.2.30 of Kotlin.
2.4.3 Objective-C

Objective-C is a superset of the C programming language [3]. There are a number of notable differences when compared to Java and Kotlin:

- Automatic Reference counting (ARC) is used instead of garbage collection. This requires a greater amount of programmer effort to avoid memory leaks.

- It is a dynamically typed language, which allows the programmer to specify the degree to which types should be specified. This introduces trade-offs between what can be expressed succinctly and what the compiler can verify. These features are supported through the Objective-C runtime library [6].

- It contains a construct called blocks which is a type of object that can capture variables and define some behavior to be carried out. These are similar to anonymous functions.

- ADTs are not a core language feature.

- Methods are fundamentally different in Objective-C, in the sense that they are represented as objects (messages) sent between objects. This is known as message passing. Any Objective-C object can theoretically receive any message, and these objects can e.g. act as proxies which relay messages which they are unable to handle to other objects.

Apple Inc [3] provide a full language reference. This report uses version 2.0 of Objective-C.

2.4.4 Swift

Swift is a language developed by Apple to replace C, C++ and Objective-C [4]. It uses the Objective-C runtime library, and as such it inherits Objective-C' memory model. In terms of features, it is more similar to Kotlin:

- ADTs are supported in the form of enums with associated values.

- It has first-class functions and closures.

- The standard library features collection types which are immutable when assigned to a constant reference.
Exhaustive case checking and pattern matching work on ADTs. Apple Inc \cite{apple2021} provides a full language reference. This report uses version 4 of Swift.

The first half of this chapter outlined the general concepts which are relevant to this thesis. The rest of this chapter will provide the specific context required for the investigation, starting with an exploration of how the Functional Reactive Programming (FRP) field evolved into the pattern which underlies Mobius and its contemporaries. The architecture introduced by the Elm programming language will be shown as it is a purely functional manifestation of this pattern. With this foundation in place, Mobius will be outlined. Finally the motivations for both Mobius specifically and this investigation in general are outlined.

\section{Functional Reactive Programming}

\subsection{Fran}

FRP's roots can largely be traced back to a library written in the Haskell programming language called Fran (short for Functional Reactive Animations) \cite{fran2021}. Fran introduces the idea of behaviors and events. Behaviors are reactive values which vary continuously over time, whereas events represent occurrences which are external to the program. These two types turn out to be isomorphic, and the term signal is therefore often used to refer to the combination of these types in later research \cite{signal2021}.

To illustrate the core concepts of FRP, Elliott and Hudak \cite{elliott2021} present the following example of creating an animation in which a circle is drawn over a square, where both shapes vary in size over time:

\begin{verbatim}
    bigger (sin time) circle \over
    bigger (cos time) square
\end{verbatim}

Here, \texttt{bigger} is a function which scales its second argument with the value of the first. \texttt{\over} is an example of an infix combinator which describes the relative positions of the objects. This expression can be translated into English as follows: Create an animation containing a circle placed over a square in which the size of the circle varies with the sine of the time, and the size of the square varies with the cosine of the time. The declarative nature of this example should be noted:
There is no coupling to the implementation details of the rendering of the animation. The circle and square are just functions of time.

### 2.5.2 Declarative Animations

Fran enables the composition of events, behaviors, and static values through the use of combinators and recursion. One key takeaway here is that Fran uses continuous time semantics. There is no notion of defining an animation frame-by-frame. The continuous time approach allows for a fundamentally higher level of abstraction than traditional imperative methods. To underline its value, Elliott [14] presents an analogy to the difference between vector graphics and bitmaps: Vector graphics can arbitrarily scale to any resolution whereas bitmaps are confined to the resolution at which they were created. In this case, vector graphics are analogous to continuous time semantics, whereas a frame-by-frame approach mirrors the bitmap representation.

### 2.5.3 Evolution of FRP

Fran’s expressiveness comes at the cost of potentially introducing space and time leaks, in addition to those already inherent to Haskell as a result of its lazy evaluation strategy [9]. In practice, continuous time semantics requires the program to poll its environment at a fixed interval, leading to unnecessary re-computation [11]. This also introduces an unavoidable latency for events which do not occur in phase with this polling period [14]. A range of follow-up FRP variants addressed these issues in various ways, such as Real Time FRP (RT-FRP) which limited the ways signals could be manipulated [34] or Event-Driven FRP (E-FRP) which use discrete time events, which was shown to be a generalizable to continuous time semantics [33].

### 2.6 The Elm Architecture

The Elm programming language, which focuses on the creation of interactive graphical user interfaces, initially took a similar approach. Its novelty to FRP was that it did not preserve the global ordering of events [11] in the interest of making applications more responsive. This asynchrony made it more difficult to reason about the behavior of programs, but was necessary for creating responsive applications.
In practice, community development in Elm morphed FRP’s concepts into a simpler pattern for structuring applications, which became known as the Elm Architecture. This new approach retained the idea of defining the system in terms of reactions to events, but replaced the complexity of creating a graph of signal combinators with a single mechanism for defining state transitions and another for effects. The idea of modelling time is also lost in this approach.

The Elm architecture consists of three basic components:

- The view
- The model
- The update function

The model describes the state of the application. The view is the user interface, which is a function of this model. The update function defines how the model is updated. It takes the program’s current state and a message describing a desired state change and returns a new state. In functional programming terms, intuitively it can be viewed as a fold function operating on the state and a stream of messages with the type:

\[
\text{update} :: \text{Message} \rightarrow \text{Model} \rightarrow \text{Model}
\]

The "Message" type is usually implemented as an algebraic data type, where each variant describes a change that should take place.

### 2.6.1 Exemplifying Elm

Czaplicki uses the example of a counter to illustrate this architecture. Here, this example is presented in Haskell as opposed to Elm in order to avoid the overhead of introducing another language’s syntax. The counter displays a number, which the user should be able to increment or decrement. The model is therefore just an integer:

```haskell
type Model = Integer
```
The set of messages describe the actions which can be taken:

```haml
data Message = Increment | Decrement
```

And finally, the update function describes how the model should change in response to these events. I.e., Increment should add to the counter, and Decrement should subtract from it:

```haskell
update :: Model -> Event -> Model
update model Increment = model + 1
update model Decrement = model - 1
```

### 2.6.2 Modelling Side Effects

The architecture can be modified to model interaction with the environment. This is done by introducing two additional types, Command and Subscription, and augmenting the type of the model function:

```haskell
update :: Message -> Model -> (Model, Command Message)
```

An overview of the Elm Architecture can be found in figure 2.1. It shows how the view (Html), as well as Commands and Subscriptions (Cmd and Sub respectively) feed messages (Msg) into the update function. The update function yields a pair containing a new model (which is fed into the subscriptions and the view) and a new command. In
addition to being purely functional, the architecture is fundamentally reactive in nature: The update function defines how the state of the application should react in response to events from various sources, and this information feeds back into the sources. As such, data flows in one direction.

2.7 Mobius

This section introduces the official version of Mobius as created by Spotify [27]. The subset which is used in this report is outlined in section 4.1. The Mobius framework draws inspiration from the Elm architecture, with some key differences. These differences are due to a mixture of embedding this pattern into an imperative language and refinements of the pattern in general.
Figure 2.2 provides an overview of Mobius’ concepts. Note the following differences between the two architectures:

- Elm’s Subscription type corresponds loosely to an ‘Event Source’ in Mobius, with the distinction that event sources are not functions of the latest model.

- The return type of the update function in Mobius includes a set of effects instead of one command. This means that the empty set can be returned if no commands are to be performed, whereas this is usually represented with a "No Operation" command in Elm.

- The implementation of effect handlers and event sources are left to the user in Mobius. This is a limitation of embedding the framework into an existing language.

- Elm provides a fully declarative abstraction over the imperative graphical user interface that underlies it [11]. Mobius does not cover this area.

- Elm’s architecture is implemented at a global application scope. Mobius can be implemented at any scope of an application, in an arbitrary number of separate instances.

- Elm has a foreign function interface for operating with potentially impure Javascript. The EffectHandler model in Mobius mirrors this, since the pure and impure parts are separated in the same way, with the distinction that both parts are written in the same language.
Chapter 3

Methods

The investigation consists of three stages, which will be described in detail in the following section:

1. **Formalization** - Outline the semantics of Mobius in a higher-level language to provide a reference point.

2. **Imperative Implementation** - Implement the framework from this reference point in the four languages, investigating alterations and their effects where appropriate.

3. **Imperative Usage** - Investigate the potential usage of these frameworks from a user's perspective, in terms of what is possible to express and what is idiomatic in the given languages.

3.1 Investigation Stages

3.1.1 Formalization

The formalization stage will translate the concepts of Mobius into a high-level functional language. Haskell was chosen for this. This is because it provides convenient abstractions for most of the concepts which need to be expressed. Also, given that the goal of this report is to find ways of programming in a more functional way, a purely functional language like Haskell makes for a good point of comparison in this regard. The results of this stage will establish the foundations needed for the subsequent stages.
3.1.2 Imperative Implementations

In this stage, Mobius will be implemented in Java, Objective-C, Swift, and Kotlin following the semantics from the first stage. Given that these are not purely functional languages and that they vary in terms of features, it is unlikely that the implementation will mirror stage 1 accurately in all regards. The goal of this stage is therefore to identify the points of contention that arise from these implementations, both with respect to the purely functional implementation and with respect to each other.

3.1.3 Imperative Usage

The final stage will investigate the usage of the framework in all four imperative languages. The goal of this stage is to determine the extent to which conforming to the desired idioms is possible and practical within the constraints that the imperative languages provide.

The stage will involve using the framework to construct the signup feature of an application. This seemingly simple example is useful, since it can involve a large amount of complexity when concurrency is required. A sign up screen should provide the user with fields for entering their name, password, and confirming their password as shown in figure 3.1. The underlying complexity of this example can be highlighted by detailing what the steps of implementing this might look like:

- First, do not allow the user to submit anything unless all input fields are filled.
- If the fields are filled but the passwords do not match, display an error to the user instead of sending a request.
- If the previous criteria are met, send the sign up request and disable the button to prevent sending multiple requests.
- Show a loading indicator once the button is pressed. Remember to remove this indicator when a response is received, or the system otherwise fails in some manner.
- If the request is successful, send the user to the application. Otherwise, enable the button to sign up again.
• If the internet connection changes, while loading or otherwise, disable the button and show an error.

• If the user starts typing again after an error has been displayed, remove the error and enable the button, unless the error was related to the user not being connected to the internet.

Figure 3.1: A simple sign up application.

Modelling this imperatively requires careful reasoning about all of these flows. The possible space for errors and the number of cases grows quickly with each added requirement. For example, the sign up button should be disabled and enabled in a number of different conditional branches. The declarative model, however, minimizes this branching.

First, the visual interface should simply be a function of the following model:

```haskell
data Model = Model { username :: String }
```
data InternetStatus
    = Unknown
    | Online
    | Offline
deriving (Show, Eq)

data SignupStatus
    = Idle
    | Loading
    | Success
    | Failure String
deriving (Show, Eq)

For example, the user interface should show a loading indicator if and only if the signupStatus field is set to Loading. The user interface does not need to know which states led up to this point or anything about the logic which determined how this field was set.

The events should model the various occurrences which can change the model. This includes all user input and the various sources of errors:

data Event
    = SignupClicked
    | UsernameChanged String
    | PasswordChanged String
    | ConfirmPasswordChanged String
    | InternetStatusChanged InternetStatus
    | SignupStatusChanged SignupStatus
deriving Show
The effect should simply model the side-effect of signing up:

```haskell
data Effect = SignUp String String
  deriving Show
```

All of the branching is collapsed into a single update function, which only needs to define transitions in terms of the previous state, instead of multiple previous states. For example, if a `SignupClicked` event is received while the model's sign up status is set to `Loading`, the event should simply be discarded, regardless of the states which led up to this happening. Note that this does not remove the inherent complexity of the underlying problem. The key is that by disregarding everything but the previous state this solution does not introduce additional complexity.

The update function (and effect handler) should be defined in terms of these ADTs. For example:

```haskell
update :: Update Model Event Effect
update model event =
  case event of
    SignupClicked ->
      -- for some signupPossible :: Model -> Boolean
      if signupPossible model
      then withModelAndEffects
        (model { SignupStatus = Loading })
        (singleton (Signup (username model)
                    (password model)))
      else noChange

    UsernameChanged -> [...] [other cases...]
```

Lines 3-11 show the case of what happens when the sign up button is clicked. If signing up is possible (as determined by `signupPossible`), the status is changed to start loading and an effect is created to indicate that a request should be sent (lines 7-10). Nothing changes if if signing up is not possible (line 11). The implementation of `signupPossible` is left out for brevity, but it should check that the fields are not blank, that the passwords match, and that the application is not in an erroneous or loading state. The other cases (lines 13-14) would be written in the same fashion.
3.2 Choice of Method

This method was chosen because it demonstrates the difficulties that multiple parties could face when using functional and reactive patterns (both users and implementers) in these languages. The combination of the implementation and usage aspects also increases the number of points of comparison between the languages, which allows the report to draw and reinforce more general conclusion. Furthermore, the implementation in a purely functional language serves to sharpen the focus of the individual implementations, and to serve as yet another point of comparison.

3.3 Evaluation Strategy

Evaluating the results of the implementations is a non-trivial task. The number of ways of quantitatively comparing the languages is limited. Where appropriate, these measurements will be taken. For instance, the size of the solutions produced in terms of number of lines or amount of files may be interesting. In addition to such metrics, this investigation will also consider and compare the concepts which can be expressed. This may include the extent e.g. the type-system or the threading model can be helpful or a hindrance in a given language.

The results section will be focused on outlining the comparisons, whereas the discussion section will address the potential implications of these results.
Chapter 4

Results

This chapter begins with the formalization of Mobius in a high-level language. Afterwards, the imperative languages are discussed individually with respect to the most important aspects of the framework. This includes analyzing what the implementation of the framework looks like in each language, and how the framework can be used. The final section provides an overview of how these languages compare to one another.

4.1 Formalization of Mobius

This section will define the semantics of Mobius which will be shared by the implementations. An instance of the Mobius framework is called a Mobius loop.

A key feature of Mobius is its ability to allow the user to specify the degree to which it should handle operations concurrently. Semantically speaking, all operations (with the exception of the updating the state) should be considered asynchronous. Given its concurrent nature, the processing order of events and effects is not deterministic.

Figure 4.1 shows how models, events and effects flow within the framework. As is shown, the environment (everything outside of Mobius) is one source of events. These events are processed in the Event Processor, which then runs the update function. The results of the update function, i.e., the new model and the set of effects, are sent to the environment and the Effect Processor respectively. The effect processor can perform side effects which can result in events being sent back to the Event Processor. These side-effects may or may not have effects...
Figure 4.1: The Components of Mobius. All arrows in this diagram represent one-way asynchronous dataflows.

The concept of channels provides a useful abstraction for defining the semantics of the components. Channels are simply a mechanism on which messages can be passed between threads or processes. Mobius requires three such channels: one for events, one for effects, and one for models.

In the following subsections, the semantics of Mobius will be outlined in Haskell. For brevity, this section leaves out some type definitions, see appendix A for the details.

### 4.1.1 Event Processing

```haskell
processEvents :: (Consumer c, Producer p) =>
  c model ->
  p event ->
  c effect ->
  model ->
  Update model event effect ->
  IO ()
processEvents modelConsumer eventProducer
effectConsumer model update =
do event <- receive eventProducer
  let (Next model' effects) = update model event
      newModel = fromMaybe model model'
    mapM_ (accept modelConsumer) model'
    mapM_ (accept effectConsumer) effects
```
The event processor starts by reading from the event channel until it receives an event (line 6). When an event is received, it runs the user's update function on it and the current model to determine if the model should change and if any effects should be carried out (lines 7-8). If the model changed, it send the new model on the model channel (line 9). It then sends each effect individually on the effect channel (line 10). Finally, it calls itself recursively with the new model if it changed, or with the old model (lines 11-12).

### 4.1.2 Effect Processing

```haskell
processEffects :: (Consumer c, Producer p) =>
    p effect -> c effect -> IO ()

processEffects effectProducer effectHandler =
    do effect <- receive effectProducer
       effectHandler `accept` effect
       processEffects effectProducer effectHandler
```

Much like the event processor, the effect processor waits for an effect to be sent to the effect channel (line 4), and when this happens it propagates this effect to the user’s effect handler (line 5), which can in turn dispatch events. Finally it calls itself recursively (line 6).

### 4.1.3 Tying it All Together

Finally, the framework’s `runLoop` function ties together these components and exposes the API to the end user.

```haskell
runLoop ::
    (Show m, Show e, Show f, WorkRunner w) =>
    m -> Update m e f -> Connectable f e ->
    w -> w -> IO (Loop m e)
runLoop model update effectConnectable
eventRunner effectRunner =
    do eventChannel <- newChan
       modelChannel <- newChan
       effectChannel <- newChan
```
Here, the user supplies the initial model, the update function, the effect handler, along with two WorkRunners which allow the user to specify which threads to run on. This function initializes the three channels (lines 7-9) and starts the effect/event processors using these arguments and channels (lines 12-18).

This function returns a loop instance which exposes a readable model channel and a function for sending events to the event channel (lines 19-21).

4.2 Formalization Implications

This high-level implementation reveals that the key abstraction here is the concept of channels, which are used to tie together the update function and effect handler supplied by the user and enable concurrency. These encapsulates the machinery of the framework, and the end-user is left with one channel for sending events and another for receiving models. With the high-level implementation as a reference, the implementations in non-functional languages can be presented.
4.3 Kotlin Implementation

4.3.1 Primitives

The primitive types in Mobius were implemented using the Kotlin interfaces as follows:

```kotlin
interface Consumer<in I> {
    fun accept(value: I)
}

interface Disposable {
    fun dispose()
}

interface Connection<in I>: Consumer<I>, Disposable

interface Connectable<in I, out O> {
    fun connect(output: Consumer<O>): Connection<I>
}

data class Next<out M, out F>(
    val model: M? = null,
    val effects: Set<F> = emptySet()
)

typealias Update<M, E, F> = (M, E) -> Next<M, F>
```

A noteworthy language feature in this regard is the ability to specify the variance of the generic arguments to these interfaces. For example, a Connectable in Mobius is contravariant in its first argument and covariant in its second.

The type declarations in the other languages will be omitted for the sake of brevity. The differences will however be described. The point of these primitives is to allow for a layer of abstraction between the components while maintaining type safety.

4.3.2 Concurrency

Much like the Haskell implementation, the Kotlin implementation provided the WorkRunner type which allows the user to define how they
wish to handle the threads that the loop runs on. As a default implementa-
tion, the ExecutorService class from Java’s standard library was
used. One of the advantages of Kotlin is that it can be compiled to the
JVM and thus interoperate with all of Java’s standard library.

Kotlin’s standard library currently contains a library for coroutines.
This library provides a range of alternative possibilities for expressing
concurrency in Mobius. For example, it contains implementations of
channels if the Haskell approach is desired. Another possibility is to
model Mobius using the Actor model of computation which this li-
brary also enables. These approaches were not explored in detail as
the library is still experimental at the time of writing this report. How-
ever, it is still noteworthy as a unique feature of Kotlin compared to
the other languages.

4.3.3 General Notes

Kotlin provides the ability to supply named arguments to functions,
which means that parameters can be supplied in any order, or skipped
entirely if a default implementation is specified. For instance, the
**Next** type with all its permutations can be defined as follows:

```kotlin
data class Next<out M, out F>(
    val model: M? = null,
    val effects: Set<F> = emptySet()
)
```

To achieve the same result in Java or Objective-C requires at least 4
separate methods. Swift is also capable of this level of expressiveness,
with the minor caveat that the order of arguments must be preserved.

4.4 Kotlin Usage

4.4.1 ADTs

ADTs can be implemented using Kotlin’s **sealed class** feature. These
are class hierarchies which must be defined in the same file. **Signup-
Event** is the superclass, and the respective events are sub-classes of
**SignupEvent**. **SignupEvent** itself cannot be instantiated.

```kotlin
sealed class SignupEvent
object SignupClicked: SignupEvent()
```
data class UsernameChanged(val name: String): SignupEvent()
data class PasswordChanged(val password: String): SignupEvent()
data class ConfirmPasswordChanged(val password: String): SignupEvent()
data class SignupStatusChanged(val status: SignupStatus): SignupEvent()
data class InternetStatusChanged(val status: InternetStatus): SignupEvent()

4.4.2 Variant Checking

Implementing variant checking for these ADTs can be done with the use of the when feature of Kotlin, which enforces exhaustive matching. As long as sealed classes are used to declare the ADTs that are being matched on, the compiler will verify that all cases are covered:

```kotlin
when (event) {
    is SignupClicked -> {...}
    is UsernameChanged -> {...}
    [other events...]
}
```

4.4.3 Representing the Model

Representing the model in Kotlin is done with use of data classes. Note that the val keyword in this case makes the reference to the value immutable. However, if said value contains mutable fields, these fields will remain mutable. It is therefore necessary to ensure all fields transitively reachable from the model are also declared with val.

```kotlin
data class SignupModel(
    val username: String = "",
    val password: String = "",
    val confirmPassword: String = "",
    val signupStatus: SignupStatus = Idle,
    val internetStatus: InternetStatus = Offline,
    val canSignup: Boolean = false
)
```
Here `SignupStatus` is defined as its own ADT and `InternetStatus` is an enum. Kotlin also provides a terse means of updating a model in an immutable way:

```kotlin
val model = SignupModel(
    username="User",
    password="Password",
    internetStatus=InternetStatus.Online,
    SignupStatus=SignupStatus.Idle
)
val model2 = model.copy(
    username="User2",
    password="Password2"
)
```

Here, `model` remains unchanged and `model2` contains a different username and password.

### 4.5 Java Implementation

#### 4.5.1 Primitives

The same approach as in Kotlin could be taken here, with the distinction that the variance of the type parameters could not be specified. These interfaces could be composed to create the composite types.

#### 4.5.2 Concurrency

Again, this approach mirrored the one taken in Kotlin.

#### 4.5.3 General Notes

In general, the Java implementation matched the Kotlin implementation. The lack of type inference and in general more verbose syntax made the Java implementation longer in terms of lines of code. The most significant differences between Java and Kotlin showed up during the usage stage.
4.6 Java Usage

4.6.1 ADTs

Like Kotlin, representing ADTs is done through inheritance hierarchies. This consists of declaring an abstract class and a set of classes inheriting from this class, which represent the variants. In comparison to Kotlin, the same ADT requires an order of magnitude more code (majority left out for brevity):

```java
abstract class SignupEvent {
    static class SignupClicked extends SignupEvent {...}
    static class UsernameChanged extends SignupEvent {
        private final String name;
        protected UsernameChanged(String name) {
            this.name = name;
        }
        @Override
        public boolean equals(Object o) {...}
        @Override
        public int hashCode() {...}
        @Override
        public String toString() {...}
    }
    [all other events as sub-classes...]
}
```

4.6.2 Variant Checking

Matching on these variants then requires checking the type of the event received. This check is not enforcibly exhaustive.

```java
if (event instanceof SignupClicked) {
    SignupClicked ev = (SignupClicked) event;
    [...]
} else if (event instanceof UsernameChanged) {
    [...]
```

\[A\] A class which cannot be instantiated
4.6.3 Representing the model

Java's `final` keyword acts as Kotlin's `val`. Using `final`, the model can be expressed as a class with the required fields, all marked as final. The status-related fields can be represented with enums. Ideally, however, the `SignupStatus` should be an ADT, but with the overhead that this would entail, it is simpler to represent the error message as its own field.

4.6.4 Meta-Programming Solutions

Most of the code presented above is repeated and predictable in structure. Annotation processing in Java can be used to deal with this overhead. The following shows an example of this using the DataEnum library [26]:

```java
@DataEnum
interface SignupEvent_dataenum {
    dataenum_case SignupClicked();
    dataenum_case UsernameChanged(String name);
    dataenum_case PasswordChanged(String password);
    [all other events...]
}
```

```java
message.match(
    SignupClicked -> {...},
    usernameChanged -> {...},
    passwordChanged -> {...},
    [all other cases...]
);
```

Likewise, meta-programming can also be used to generate the class used for the model in a similarly terse manner.
4.7 Swift Implementation

One of the main difficulties in the Swift implementation was related to the use of generic types. Mobius is made up of compositions of simple types such as Consumer or Disposable. For example, the Connection type is the composition of these two types. These types are often parameterized by a generic type, which can be expressed using interfaces in Kotlin and Java. The appropriate representation in Swift is less clear. This report investigated the following alternatives.

4.7.1 Protocol with an Associated Type

```swift
protocol Consumer {
    associatedtype InputType

    func accept(_ value: InputType)
}
```

This option means that classes can be created to conform to the protocol. For example, `Connection` can implement it as follows:

```swift
struct Connection<Type>: Consumer {
    typealias InputType = Type
    [...]}
```

However, this does not allow for the declarations such as:

```swift
let consumer: Consumer<SomeType>
```

This is because, as stated, protocols do not take types as arguments.

One solution is to use a pattern known as type-erasure. This involves creating another class, which can take any connection as an argument and dispatch to it:

```swift
protocol Connection: Consumer, Disposable {}

class AnyConnection<Input>: Connection {
    let acceptFn: (Input) -> Void
    let disposeFn: () -> Void
```
To explain, `AnyConnection` takes a user-defined implementation of the `Connection` protocol and its associated type (lines 8-13). When `accept` and `dispose` are called on the `AnyConnection` object, it defers these calls to the user-supplied `Connection` object (lines 16 and 20).

Fields of type `AnyConnection<SomeType>` can then be declared, and be instantiated like this:

```swift
class MyConnection<Type>: Connection {...}

let connection: AnyConnection<Type> = AnyConnection(MyConnection())
```

This is known as type erasure since the `AnyConnection` wrapper *erases* the underlying type supplied to it. This approach preserves the desired property that the types should be composable, at the cost of introducing these extra type erasure classes.
4.7.2 Typealiasing

An alternative approach to Protocols with associated types is to use typealiasing, i.e. creating a type-parameterized alias which models the primitive types. Since a consumer is determined completely by its `accept` method, the following alias could be used to model it:

```swift
 typealias Consumer<Value> = (Value) -> ()
```

I.e., a consumer is a function from `Value` to the empty type. This enables values to be declared to be of type `Consumer<SomeType>`, but does not allow, for example, `Connection` to declare itself to be a consumer, since a class cannot conform to a function type in Swift. This removes the ability to compose these types.

4.7.3 Emulating Abstract Classes

A third approach is to emulate the idea of abstract classes. Instead, a subclass must be created which implements the functionality defined in the abstract class. Swift does not directly support this concept, but it is possible to emulate with runtime checks. `Consumer` could be defined in this manner:

```swift
open class Consumer<Type> {
    func accept(_ i: Type) {
        fatalError("This class must be sub-classed")
    }
}
```

There are two disadvantages of this approach. The first is that the compiler is not able to verify that the class will not be instantiated. The second is that this method also does not compose in the desired way, since Swift does not support multiple inheritance.

4.7.4 Primitive Implementation and Implications

Out of these three alternatives, using protocols with associated types and type erasure wrapper classes was deemed the most appropriate since it was the only one which preserved the desired property of composing the types. Although this approach is more verbose than the approaches taken in Java and Kotlin, the overhead is largely covered
by the implementer of the framework. An end user is still able to use the APIs in largely the same way, and can supply implementations of these types using higher-order functions instead of subtyping. The cost would only be noticeable in the rare case that the user needs to extend these primitives and add other generic functionality.

4.7.5 Concurrency

The WorkRunner type from the Haskell implementation was implemented with a Swift protocol. This protocol exposed one method which took an anonymous function. A concrete implementation was also created which used Swift’s DispatchQueue API. This abstraction enables actions to be added to a queue which are handled by an underlying thread pool.

4.8 Swift Usage

4.8.1 ADTs

Swift provides ADTs in the form of enums with associated values. Using this, the login events can be expressed as follows:

```swift
enum SignupEvent: Hashable {
    case loginClicked
    case usernameChanged(username: String)
    case passwordChanged(password: String)
    [...]}
```

4.8.2 Variant Checking

Variant checking is similarly terse. Out of the imperative languages, Swift is unique in this regard as it allows pattern matching in these expressions:

```swift
switch event {
    case .signupClicked: [...] 
    case .usernameChanged(let username): [...] 
    case .passwordChanged(let password): [...] 
```
4.8.3 Representing The Model

Representing the model in Swift largely mirrors the approach taken in Kotlin. In this instance, the `struct` construct was used because they are passed by value (whereas structs are passed by reference). The `let` keyword is used exclusively to signal that the fields are immutable.

```swift
struct SignupModel: Equatable {
    let username: String
    let password: String
    [...]
}
```

Finally, a means of updating the model is needed. Default parameters can be used to emulate the `copy` method used in Kotlin:

```swift
struct SignupModel: Equatable {
    [...]
    func copy(
        username: String? = nil,
        password: String? = nil,
        [...]
    ) -> SignupModel {
        return SignupModel(
            username: username ?? self.username,
            password: password ?? self.password
            [...]
        )
    }
}
```

4.9 Objective-C Implementation

4.9.1 Primitives

Like Swift, Objective-C's type system poses challenges for implementing Mobius in a type-safe fashion. However, given that Objective-C
is a dynamically typed language, creating a completely type-safe implementation is not necessarily idiomatic or desirable. For this reason, this report's Objective-C implementation took a largely dynamically typed approach. The primitives were declared using protocols, and the threading model used the same APIs as the Swift implementation.

Unlike Swift, Objective-C does not provide a terse means of creating ADTs. Therefore, this investigation took advantage of Objective-C's more dynamic type system and message passing system to take an alternative approach to the same problem.

### 4.9.2 ADT and Message Passing Correspondence

As stated, Objective-C methods are actually just data sent as messages between objects. The programmer can choose to arbitrarily intercept and forward these messages between objects without the caller seeing this.

In Objective-C, a *protocol* defines a set of messages that an object should implement in order to conform to the protocol. If these messages are viewed as the variants of an ADT, then an object which conforms to this protocol is like a function which can perform some action for each of the variants. Also, each message can hold data, just like each variant of an ADT. Figure 4.2 demonstrates this correspondence.

![Figure 4.2: A representation of events using ADTs and a corresponding representation as an Objective-C protocol](image)

With this correspondence, the Mobius update function can be implemented in terms of a protocol. The user supplies the loop with an object conforming to the protocol, and sends the protocol's messages
to the loop directly. The loop re-routes these messages internally to the user-supplied object on the desired thread. The effect handler can be represented in a similar fashion. The distinction here is that the \textbf{Next} object returned by the update function should contain a set of messages.

One issue with this approach is figuring out where the model should be stored. In order for the update protocol to be functionally pure, the model must be supplied as an argument to each of its messages. However, this requires the user to explicitly send the latest model along with each event. Functional currying can be used to address this issue: the events are represented as higher-order functions which take any arguments to the event and return a function which takes the model which the framework can then call with its internally held current model. This augmented type is represented by the \texttt{UpdateFn} type alias in figure 4.2.

The effect handler has a similar issue that is solvable in the same way. The problem is that it requires a reference to the loop’s event channel. The messages in this protocol should therefore take the effect’s arguments and return a function of the event channel. The user supplies in turn supplies the effect’s argument and the framework performs the effect by supplying the event channel.

Although this approach is not enforcibly type-safe without external tools, it does allow the user to create a functionally pure and easily testable API. This removes the need for ADTs. It also removes the need for exhaustive variant checking since the user must handle all cases by the definition of a protocol. In terms of the implementation, there is some overhead to this approach compared to a purely data-centric approach. However, this cost can largely be covered on the implementation side and as such does not imply any significant overhead when using the framework.

This report did not investigate if protocols can be used to encode the self-recursive property of ADTs. Without this, the two constructs are not isomorphic, i.e., arbitrary ADTs cannot be represented with protocols. This was not a problem in this investigation, since this property was not used.
4.9.3 Concurrency

Objective-C and Swift use the same runtime and have access to the same threading library (although the Swift standard library provides a marginally higher abstraction for using it). This meant that threading could be implemented in much the same way as Swift, with the same results.

4.9.4 General Notes

Even after refactoring, the Objective-C implementation ended up being at least twice as large as any of the other implementations in terms of the number of lines of code. Much of this can be attributed to the language’s requirement of splitting all classes into header and implementation files, as well as the language just generally being relatively more verbose when expressing most concepts. See section 4.11 for a more detailed discussion about the implications of the difference in size.

The lack of type-safety in the Objective-C implementation is also noteworthy. Essentially all type-safety which related to generics was lost as well as the ability to check that the update and effect handling protocols conform to the expected structure. This report does not seek to comment on whether this dynamism is to be considered a good or a bad thing, simply to say that this is a reflection of the differences between the languages.

4.10 Objective-C Usage

4.10.1 ADTs and Variant Checking

The correspondence between ADTs and message passing enables the events and effects to be defined tersely. E.g. the events were defined as follows:

```markdown
@protocol SignupEvent <NSObject>

typedef id <SignupModel> SignupModel;
typedef Next<id <SignupModel>, id <SignupEffect>>
    *(^Updater)(SignupModel);
```
Implementing variant checking is then simply a matter of implementing this protocol in a class.

### 4.10.2 Representing the Model

In contrast, the implementation of the model was more verbose. The chosen method involved creating a protocol which exposed the model's fields and a set of functions for updating these fields with new models. Objective-C supports the ability to declare fields as immutable with its `readonly` annotation. This was used exclusively (lines 6-9):

```objective-c
@protocol SignupModel <NSObject>

enum InternetStatus {...};
enum SignupStatus {...};

@property (readonly) NSString *username;
@property (readonly) NSString *password;
[other fields...]
@property (readonly) NSString *errorMessage;

+ (id <SignupModel>)defaultModel;
- (id <SignupModel>)withUsername:(NSString *)username;
- (id <SignupModel>)withPassword:(NSString *)password;
[constructors for all other fields...]
@end
```
A class was then created which conformed to this protocol, which implemented each of the with* methods (e.g. withUsername, withPassword, etc) by calling the classes’ constructor again with the changed parameters. E.g., the method which changed the model’s username was implemented as follows (the variables prefixed with '_' are the classes current internally held fields):

```objective-c
-(id <SignupModel>)
  withUsername:(NSString *)username {
    return [[SignupModelImpl alloc]
      initWithUsername:username
      password:_password
      internetStatus:_internetStatus
      [other fields...]];
  }
```

The important thing to note here is that there is no construct in the language which prevents the underlying representation of the model protocol from being internally mutable. Also, this approach does not allow one to change several fields at once like Kotlin’s copy function.

The implementation of the model is also difficult to change. If a field is to be added, this requires two changes to the protocol (i.e. adding a @property and a with* method for updating this property) and a number of changes which scales linearly with the number of fields in the model (since each constructor invocation needs to be altered).
4.11 Summary of Results

Table 4.1 depicts an overview of the support that these languages provide for the most relevant features. In this figure, *Not enforceable* means that the property cannot be statically verified, and is left up to individual programmer discipline.

None of the implementations or usages were optimized for the size. Table 4.1 only contains line numbers as a relative measure between the languages. In the table, *Core Components* refers to the core mechanics of the framework. This is what would need to be understood in order to grasp these mechanics.

The total implementation sizes in Swift and Objective-C were inflated due to the type-erasure pattern described in 4.7.1. For the sake of practicality, this would probably be left out entirely in the Objective-C implementation given its already limited type safety. These types therefore do not provide much additional utility in Objective-C.

The usage size in Java was inflated considerably by not using metaprogramming. Roughly 250 lines could likely be removed if metaprogramming were used, albeit with the additional overhead of code generation libraries themselves.

As a general conclusion, if the size of the solution is deemed a relevant metric, Objective-C was the only outlier in this regard on the implementation side. Java was an outlier on the usage side, but notably due to the lack of meta-programming.
Table 4.1: A summary of the features supported by the 4 languages

<table>
<thead>
<tr>
<th>Feature</th>
<th>Java</th>
<th>Kotlin</th>
<th>Swift</th>
<th>Objective-C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algebraic Data Types</strong></td>
<td>No. Possible with code generation</td>
<td>Yes</td>
<td>Yes</td>
<td>Not directly. Can be emulated with protocols</td>
</tr>
<tr>
<td><strong>Variant Checking</strong></td>
<td>No. Possible with code generation</td>
<td>Yes</td>
<td>Yes</td>
<td>Not directly. Can be emulated with protocols</td>
</tr>
<tr>
<td><strong>Immutable Data Objects</strong></td>
<td>Not enforceable. Impractical without code generation</td>
<td>Not enforceable. Possible with data classes</td>
<td>Not enforceable. Possible with structs.</td>
<td>Not enforceable. Impractical</td>
</tr>
<tr>
<td><strong>Type Safety</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Limited</td>
</tr>
<tr>
<td><strong>Anonymous Functions</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Referential Transparency</strong></td>
<td>Not enforceable</td>
<td>Not enforceable</td>
<td>Not enforceable</td>
<td>Not enforceable</td>
</tr>
<tr>
<td><strong>Concurrency</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Lines of code (Entire Implementation)</strong></td>
<td>289</td>
<td>204</td>
<td>310</td>
<td>747</td>
</tr>
<tr>
<td><strong>Lines of code (Core Components)</strong></td>
<td>138</td>
<td>112</td>
<td>153</td>
<td>271</td>
</tr>
<tr>
<td><strong>Lines of code (Usage)</strong></td>
<td>451</td>
<td>124</td>
<td>178</td>
<td>346</td>
</tr>
</tbody>
</table>
Chapter 5

Discussion And Evaluation

The results outline two recurring themes. First, that a missing language feature can often be mimicked through the use of other language features. And second, that these languages are equally limited by their lack of ability to verify immutability and referential transparency.

5.1 Language Feature Replacements

The results section shows several examples of the flexibility of these languages when it comes to replacing or emulating missing features. For example, it was shown that ADTs can be added to Java with annotation processing, and replaced in Objective-C by drawing a parallel to its message passing system and protocols. This insight highlights the difficulty in evaluating a language based on its promoted set of features alone. There are likely non-trivial solutions that can be used to implement, or at least approximate, a desired pattern or feature.

However, such replacements have implications that are worth considering. For example, the approach taken in Objective-C is useful, but also unorthodox. Similarly, meta-programming can require a large amount of extra code to be imported, likely from third-party sources. In addition to potentially introducing faults, this can also have adverse effects on compilation times.
5.2 Functional Programming Approximations

Immutable data is a cornerstone of functional programming. Mobius operates under the assumption that its model is immutable. Kotlin and Swift stand out in this regard, as they both provide terse means of creating and modifying immutable objects. However, all of the languages fall short in the sense that they cannot guarantee immutability, nor do they make it idiomatic. The developer needs to know, for example, to mark all fields in their model with some form of readonly modifier, and to also do so transitively to anything referenced by the model. This is not only error-prone, but may often not be possible if the user needs to refer to types created by a third-party.

Referential transparency is another cornerstone of functional programming, and again, these languages fall short in this regard. There is nothing to prevent the user from performing arbitrary side-effects in the update function, for example.

These issues can be addressed in a number of ways. For instance, mutation checks could be performed during runtime by the framework in a debugging-mode. Static analysis is another potential solution, which could potentially catch other side-effects as well. However, these are unlikely to be able to catch all violations.

These types of problems highlight the pain points of approximating functional programming in imperative languages. Immutability and referential transparency form the foundations of the functional paradigm, and given that neither principle can be enforced in these languages, that foundation is inherently shaky. If this endeavour is to be pursued, it is likely that the commonly held idioms of these languages need to be rethought.

5.3 Threats to Validity

There are several factors which have implications for the results of this report:

- **Implementation details** - There may exist better ways of implementing the features than the approaches taken in this investigation. The goal is not to be prescriptive. See section 5.1 for a more detailed discussion of these implications.
• **Choice of languages** - Investigating all common programming languages would be outside the scope of this investigation. These languages were chosen because they balance the time taken to perform in-depth investigations of each language and the value of investigating a breadth of languages. See section 2.4.

• **Choice of Framework** Mobius was chosen as a basis for this investigation as it is largely based on the patterns from the fields of functional and reactive programming. These represent ideas which are grounded in theory and also prevalent in industry.
Chapter 6

Related Work

In addition to Elm, there are a number of efforts which relate to our investigation. This section outlines the main ones which were investigated.

6.1 Frappé

Frappé is a framework which embeds FRP in Java and is based on a correspondence between FRP and the Java Beans programming model [8]. A Java Bean is a Java class which adheres to a set of conventions, the most relevant of which is the idea of bound properties. A bound property is a field which a client can subscribe to in order to be notified of any changes to that field. The authors map this model to the idea of components reacting to changing signals in FRP.

Frappé uses a graph to represent an FRP program [8]. Nodes in this graph are FRP combinators (implemented as classes) and edges are fields which reference additional combinators. A parallel can be drawn between this representation and the structure of Mobius. The update function could be modelled by a set of nodes, where each node represents a different Mobius Event. The environment, Effect Handler, and Event Handler would feed events into the source of this graph, and the sink would be the mechanism which sends new models to the environment and effects to the effect handler. Edges in this graph would represent one-way asynchronous calls.
6.2 Flapjax

Flapjax is both a Javascript library and a language for FRP \cite{23}. It is similar to Frappé in the sense that it also constructs programs as graphs. The sources in this graph include, for example, user input and clocks whereas the sinks can be, for example, the user's screen or the network. The sources push values into the graph at discrete time intervals, which propagate through the nodes in the graph. Each node can apply a function to its value and propagate the result to its child nodes. Nodes may have several sources, and therefore act as FRP combinators.

6.3 Redux

Redux is a JavaScript library which also draws inspiration from Elm \cite{2}. Like Mobius, it relies on a uni-directional flow of data through update functions (called reducers in Redux). Events in Mobius correspond directly to Actions in Redux.Reducers have the type:

\[
\text{reduce} :: (\text{State}, \text{Action}) \rightarrow \text{State}
\]

In contrast to Mobius, Redux leaves it up to the programmer to decide how and where effects should be handled in the chain. Recreations of Redux exist in various languages, but no study into comparing language implementations could be found by this investigation.
Chapter 7

Conclusion

This thesis posed the following research question:

How can certain language features of imperative languages enable or hinder programming in a functional, reactive and declarative style?

This question was answered through the specification, implementation and usage of a framework called Mobius, which incorporates ideas from these three areas. The investigation centered around four programming languages: Java, Kotlin, Swift, and Objective-C.

The results show a range of features which proved useful and the degree to which each language could express these tersely (see 4.11). In the cases where expressing a concept proved impractical, alternative solutions were explored. For example, a terse notation for algebraic data types can be added to Java through the use of annotation processing.

Therefore, the difficulty does not necessarily lie in the ability to express the concepts, but in the extent to which doing so is idiomatic. This is especially important given that these languages do not provide means of verifying two central properties of functional programming: immutability and referential transparency.

This investigation operated under the hypothesis that Kotlin and Swift would stand out in terms of their features compared to the other languages. However, the results show that the only noteworthy difference was in terms of the ability to express algebraic data types. Java and Objective-C are both capable of replacing the need for this feature in their own ways. The fact that the overhead for these replacements
can be placed on the implementation side means that this difference is minimal in practice. The hypothesis has therefore been shown to be incorrect.

7.1 Future Work

This investigation has addressed one approach to functional and reactive programming in a small set of languages. While the results can be useful, they would be strengthened by investigating more languages from more paradigms. Furthermore, state management is just one part of programming in this style. One could e.g. investigate ways of building UIs declaratively in different languages.

The field of static analysis could also be explored with regards to these issues. For instance, there may be ways of statically detecting when the user has failed to write referentially transparent functions in parts of the program which should be pure (such as the update function).

Finally, the issue of performance is important to address for real-world usage of the framework. As was mentioned in the introduction, the Mobius model of programming and functional/reactive programming in general lend themselves well to interactive systems. If an implementation does not perform well, it may disqualify it from consideration even if it fulfills all the other requirements.
Bibliography


Appendix A

Haskell Mobius Type Definitions

The following provides the type definitions used in the Haskell implementation of Mobius:

```haskell
import Data.Set as Set
import Control.Concurrent

data Next model effect =
    Next (Maybe model) (Set effect)

type Update model event effect =
    model -> event -> Next model effect

type Connectable i o = Chan o -> i -> IO ()

class Consumer c where accept :: c a -> a -> IO ()

class Producer p where receive :: p a -> IO a

class WorkRunner runner where
    run :: runner -> IO () -> IO ()

data Loop model event =
    Loop { observe :: Chan model
            , dispatch :: event -> IO ()
            }

-- Implementations
```
instance Consumer Chan where
  accept = writeChan

instance Producer Chan where
  receive = readChan

data EffectHandler effect event =
  EffectHandler (effect -> event -> IO ())

data Fun a = Fun (a -> IO ())

consumerFromFunction :: (a -> IO ()) -> Fun a
consumerFromFunction = Fun

instance Consumer Fun where
  accept (Fun f) value = f value