Bachelor thesis
Computing Science

Radboud University

Cross-Platform Mobile App Development

Translating SwiftUI (iOS) to Jetpack Compose (Android) and vice versa

Author:
Toine Hulshof
s1005649

First supervisor/assessor:
dr. J.E.W. Smetsers
s.smetsers@cs.ru.nl

Second assessor:
dr. P.W.M. Koopman
pieter@cs.ru.nl

June 26, 2020

Abstract

Cross-platform development for mobile applications is interesting for the software development industry, because a single code base reduces development time and effort, is easier to maintain and makes an app available to more people. We present two programs that translate user interfaces; one for translating SwiftUI (iOS) code to Jetpack Compose (Android) and one translating vice versa. We show the results of these programs on two example apps. The final product supports developers in creating cross-platform mobile applications, but does require manual post-processing of the translated code as, among other things, platform specific APIs are not feasible to convert. Our programs are recommended for developers who prefer to write native code on smaller projects, however for production use, the use cross-platform alternatives is advised.

Contents

Chapter 1

Introduction

1.1 Problem Description

The original iPhone, introduced by Steve Jobs on the 29^{th} of June 2007 marks the beginning of the global smartphone revolution. Ever since that historical date, over 13 billion smartphones were sold and as of 2020, 3.5 billion people have one available right at their fingertips.
These smartphones would not been considered that smart nor would they have been so popular without the introduction of the Apple App Store (July 10^{th} 2008) and the Google Play Store (October 22^{nd} 2008). The online stores for mobile applications contain over 2.2 million and 2.8 million apps respectively. Since these apps are a modern gateway to financial success and a large userbase, it is essential for businesses to step on this mobile application train.
To develop a cross-platform mobile app, one would have to code and maintain two different codebases. The default programming language for iOS was `Objective C` and for Android, this language was `Java`. These different codebases, would cost businesses significantly more time and money to develop and maintain, than when just a single codebase was used. Behold the arrival of the cross-platform tools. These tools would help developers to develop an app for multiple platforms in less time with just a single codebase. There were several drawbacks with these tools, but at that time it was worth the convenience of developing an app just once. One of these drawbacks is that the native APIs and frameworks would not work on both iOS and Android.
It wasn't until the cross-platform development tools React Native[1] (Facebook, 2015) and Flutter[2] (Google, 2018) that these limitations were resolved. An application written with these frameworks could be compiled to native iOS and Android source code, which greatly increased performance and

[1] React Native, Facebook 2015. `https://reactnative.dev`
[2] Flutter, Google 2018. `https://flutter.dev`

functionality. While React Native and Flutter are said to be great tools, it does not give developers the feeling and recognition of writing a native application for either iOS or Android. This might soon change.
On June 3^{rd} 2019, Apple announced at their yearly developer conference the framework SwiftUI[4]. This framework builds on the then 5 year old programming language Swift[2], also created by Apple. SwiftUI ensures that user interfaces (UIs) are not created imperatively anymore, but rather declarative. In short this means easier to write code, faster to develop and a smaller codebase. According to the majority of iOS developers[11], this newly developed framework is the future of creating user interfaces.
This statement is reinforced by the fact that Google released the framework Jetpack Compose[8] on October 23^{rd} 2019; not much later than SwiftUI. Jetpack Compose builds on the programming language Kotlin[7] and has the same declarative approach to developing UI as SwiftUI.
Since SwiftUI and Jetpack Compose consist of shared features, similar data flow and similar notation, we figured it would be possible to write a translator from SwiftUI to Jetpack Compose and vice versa. For these reasons we developed a sample version of such a translator. After this implementation is done, the pros and cons will be considered and will be compared to the currently available cross-platform tools. Therefore the question that will be answered in this thesis is:

To what extend can a SwiftUI to Jetpack Compose translator (and vice versa) be developed and how does this implementation compare to the currently available cross-platform mobile development tools?

1.2 Chapters Overview

Chapter 2 will give a deeper insight in what cross-platform developer tools are currently available. After that, in Chapter 3, the goal and motivation of our research is formulated. This chapter also includes an overview of our proposed solution. Chapter 4 includes the reasoning behind our implementation decisions. We go over different methods we considered to implement our translation tool. This is followed by a more in depth explanation of why and how our implementation works. In Chapter 5, we test our implementation on one iOS and one Android mobile application. We go over the post-processing steps that are required to make compilable code that generates UI which resembles the original UI. Chapter 6 includes the general limitations of our programs and concludes in what scenario our programs should be considered to being used. Finally, in Chapter 7 we present ideas for future work regarding the usability of our product and how the product could be improved.

Chapter 2

Background

2.1 Frameworks

First, we explain how the frameworks SwiftUI and Jetpack Compose work. Both frameworks work in essence the same; meaning that the declarative UI design and data-flow principles occur in both frameworks.

2.1.1 UI Tree

Building a user interface using any of the two frameworks is declarative, as has been said. This means that rather than telling the screen renderer how a UI should be created (imperative), we tell how a UI should look and the screen renderer figures out how to achieve this.
This is achieved by declaring a single view as the root view of the program. This view can contain multiple children views (for example stacked underneath each other).
Rather than defining the entire UI in a single view, in both frameworks we have the possibility to split this view into smaller subviews (also called widgets). This also allows us to re-use these views.
When an application is being built, the compiler creates the entire UI Tree. This tree is created by replacing the root widget with the content this widget refers to. The widgets in this content are thereafter recursively replaced by the content *that* widget refers to. This process finishes when all the widgets are replaced with the fundamental widgets (e.g. `Text`, `Button` or `Image`). The UI tree is now transformed into a view. This view will then be displayed on the screen. An example of a view defined in SwiftUI is displayed in Figure 3.1 and a Jetpack compose example is displayed in Figure 3.2.

2.1.2 MVVM

The other important principle of these frameworks is the way how data flows. SwiftUI and Jetpack Compose heavily depend on the Model-View-ViewModel (MVVM) design pattern.
An application designed using MVVM contains three elements.

1. The **Model**. This model contains the data that the application should represent.

2. The **View**. The view should, given the model, define how the UI should look.

3. The **View-Model**. The view-model allows for the communication between the model and the view. Whenever the model changes, a publisher inside the view-model is triggered. The view-model then recalculates which parts of the view must be updated in order to be consistent with the model.

In contrary to the Model-View-Controller (MVC) design pattern, MVVM allows for two-way binded data[10]. This supports the idea of a single source of truth[5], which implies that every bit of data is declared only once. This principle ensures that data only lives in one place and that consistency between the state (data) and the view is preserved at all times. This eliminates inconsistencies between the model and the view, so the view is always up to date with the current model thanks to the view-model.

2.2 Cross-platform Alternatives

There are multiple ways to have a single codebase work on both iOS and Android simultaneously. Over the years, more and more advanced development tools have been created to make it easier for developers to distribute their ideas in the form of a mobile application. 99.32% of mobile users worldwide are reached with a cross-platform application, while a native iOS or Android app would only reach 26.8% and 72.52% of the mobile users in the world respectively[1]
According K. Shah and H. Sinha[9], we can divide these cross-platform tools into three categories; web apps, interpreted apps and widget based apps. For each of these categories, we will state an example framework of this category, its features and limitations, as well as the pros and cons.

[1]StatCounter, 2020. Accessed at `https://gs.statcounter.com/os-market-share/mobile/worldwide` on 09-06-2020

2.2.1 Web Apps

A web app uses an internet browser to retrieve and display data. The code runs on a server instead of the device.

Pros

- Web apps can run on any device that has an internet connection. Instead of using the browser for such an app on a mobile device, it is possible to create an application for either iOS or Android that basically renders a webpage.
- It is not required that the software is downloaded to the device. However, when the web app is displayed within an application, a download is required.

Cons

- The hardware of the mobile devices cannot be used to their maximum potential. This is because the application runs in the browser and the code runs on the server, so it cannot utilize the CPU nor the RAM of the device directly.
- The user always requires an internet connection in order to use the application.
- Platform specific APIs are difficult to implement and whenever an API is available on multiple platforms, a single solution is used. This means that operations must work on any platform which could lead to a decrease in performance, since the operations are not optimized for the targeted device.

Example

Popular frameworks in which web apps can be created are Angular[2], created by Google which uses `HTML`, `CSS` and `JavaScript` and Django[3], which uses `Python`. These frameworks are usually used to create a web application that is initially only intended for the web. When demand for a mobile application and the resources of the developers is low, a web app as a temporary substitute for a native mobile application is a viable option.

2.2.2 Interpreted Apps

An interpreted app is an application that deploys the source code to the mobile device, after which a JavaScript engine interprets this code and displays the result on the screen.

[2]Angular, Google 2010. `https://angular.io`

[3]Django, Django Software Foundation 2005. `https://www.djangoproject.com`

Pros

- The main pro of this category of cross-platform applications is that native apps can be created using simple JavaScript.
- Because these apps are native, hardware can be accelerated which results in better performance.

Cons

- In an interpreted app, UI is written in JavaScript. It is then compiled to the native platforms. JavaScript is not statically typed, which could lead to unforeseen bugs and crashes which could have been detected at compile-time.
  This issue can be resolved by using `TypeScript` as the scripting language for a interpreted app when this option is available.
- Like web apps, interpreted apps also do not offer platform specific APIs.

Example

The most well-known example of an interpreted app is React Native[4]. This framework is created by Facebook and build on ReactJS to ensure that code written with React, it can be deployed on any platform; including mobile operating systems.

2.2.3 Widget Based Apps

A widget based app is an application that treats every element as a widget. These widgets combined form a UI Tree, which is discussed in Section 2.1.1. Widget based apps follow the same principle as SwiftUI and Jetpack Compose. The components that follow from the widgets are dynamically created using a rendering engine that converts the application code into native code.

Pros

- Widget based apps usually use the `MVVM` (Section 2.1.2) design pattern for data flow. This means that there is a single source of truth, which when updated causes the view to update without additional work from the developer.
- Since SwiftUI and Jetpack compose are native widget based apps, another widget based framework is familiar to a native app developer. This makes it easier for the developer to switch from framework and adhere to the design principles of such a widget based app.

[4]React Native, Facebook 2015. `https://reactnative.dev`

Cons

- Like the other mentioned application categories, it is difficult to make platform specific APIs work and compile on iOS and Android.
- A widget based app must be able to infer how the UI should look on both mobile operating systems, which results in at least the same amount of code in theory, but in practice to a lot more verbose code. This could reduce the readability and maintainability of the code.

Example

The most well-known widget based app is Flutter[6]. In 2017, Google used part of its resources to develop Flutter. The code is written in the language `dart`. Dart creates the native components from the widgets using the Skia 2D graphics engine[9].
Flutter becomes more and more popular nowadays with respect to React Native. Even though React Native offers better support for platform specific APIs since it has matured over the last 5 years, Flutters ease of use and excellent documentation[5] makes it more appealing for developers as a cross-platform tool. To add on to this, a reason to choose React Native over Flutter is that React Native supports web applications while Flutter for the web is currently under development. Besides these differences, Flutter and React Native have a lot of aspects in common like native looking UI elements. For example, a button defined in either framework matches the appearance of a native button in iOS or Android.
Although Flutter does a lot of things right, it cannot be denied that beginner and experienced developers prefer to write in the language that was made specifically for the purpose of writing applications for their preferred mobile operating system.

[5]Flutter vs React Native, B. Skuza, A. Mroczkowska, D. Włodarczyk. `https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2020`

Chapter 3

Goal

3.1 Motivation

The problem of writing a reliable, well-functioning mobile application that works on the two largest mobile operating systems, Android and iOS, is huge. Businesses have to hire two separate engineering teams, make sure that both apps are similar in user interface and features and both code bases must be maintained. To be able to develop an app just once has numerous advantages. Namely, the developer can develop in their favorite environment, it reduces initial costs, increases productivity and is easier and cheaper to maintain. This problem is already partly solved by the currently available tools, however this does not give the familiarity and ease of use of the coding environment an iOS or Android developer is used to. It is also more cumbersome to write an app in a completely different language which is lesser known and more verbose, as it requires additional boilerplate code.

3.2 Final Product

After this research, we have accomplished the following goals:

- We developed two applications that translates a SwiftUI project to a Jetpack Compose project and vice versa. There is however manual post-processing required to make the code compilable and work as expected.
- We tested the implementation of both programs on example applications of both mobile operating systems. We described the post-processing steps required to make the application compilable. We also show some screenshots to visualize the results.
- We compared our programs to currently available cross-platform tools and listed the pros and cons of each of them. We described for what

developer and in which situation our program can be used to accelerate cross-platform mobile development.

3.3 Method

To conduct this research, the following steps were taken:

1. We gained insight in the grammar of SwiftUI and Jetpack Compose.
2. We developed the SwiftUI to Jetpack Compose translator as complete as possible with the insights we had at that moment in time. We tested the program on a wide range of SwiftUI views. When everyone of these views would translate correctly and as complete as possible, the program was finished. Otherwise we gained more insights about the characteristics of the frameworks to improve the programs.
3. We developed the Jetpack Compose to SwiftUI translator following the same steps as the SwiftUI to Jetpack Compose translator. We made the translator as complete as possible with the insights we had at that moment in time. Thereafter we tested the program on a wide range of Jetpack Composes composable views. When everyone of these views would translate correctly and as complete as possible, the program was finished. Otherwise we gained more insights about the characteristics of the frameworks to improve the programs.
4. We tested each implementation on an example app. We manually post-processed the result of the generated applications to make it compilable and work as exprected. We listed the adjustments we made.
5. We compared our implementations against alternative cross-platform development tools and listed the pros and cons.

3.4 Intuition behind our Programs

The intuition behind our solution comes from the fact that the recently released frameworks show significant similarities. Both frameworks pushed the industry from imperatively defining the user interface to a declarative manner. We considered it would be possible to write a translator, which among other things, would preserve the ease-of-use, readability and data flow. The ideal solution would be to transfer an entire application written within the preferred operating system of the developer an to the other mobile operating system with a push of a button.
We illustrate the similarities of the frameworkds based on a simple example. This example shows how a simple counter application should be created in SwiftUI and Jetpack Compose.

```swift
struct Counter: View {

    @State private var count = 0

    var body: some View {
        VStack {
            Button("Press me") {
                self.count += 1
            }
            Text("I have been pressed \(count) times.")
                .foregroundColor(.red)
        }
    }

}
```

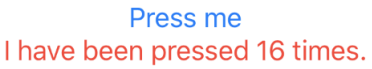


Figure 3.1: Example SwiftUI implementation for a counter application and its rendered result.

```kotlin
@Composable
fun Counter() {
    var count = +state{0}

    Column(modifier = Modifier.fillMaxSize(), arrangement =
    ↪ Arrangement.Center) {
        Button(text = "Press me", onClick = {
            count.value++
        })
        Text(
            text = "I have been pressed ${count.value} times.",
            style = TextStyle(color = Color.Red)
        )
    }

}
```

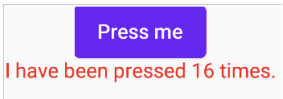


Figure 3.2: Example Jetpack Compose implementation for a counter application and its rendered result.

We noticed the following things.

1. It can be derived from the keyword `View` or the annotation `@Composable`

that we are dealing with a UI element.

2. Both frameworks have an easy way to handle state. Whenever a variable is marked with `@State` in SwiftUI or with `+state` in Jetpack Compose, the view automatically re-renders when this variable changes to reflect the new state.

3. As both frameworks use declarative UI design, it can be seen that a `VStack` simply translates to a `Column`. There is a small catch however; Jetpack Compose renders views automatically from the topleft corner while SwiftUI renders from the center of the screen. This is the reason that the `Column` requires arguments to place it in the middle of the screen.

4. A modifier in SwiftUI, for example the `foregroundColor()`, should be translated to an argument of `Text` in Jetpack Compose. This is possible, but we had to keep this in mind while developing the translator. The fundamentals of both frameworks can be found in Section 2.1

The more technical and in depth explanation of how and why our proposed solution works, can be found in Chapter 4.

3.5 Comparison

There has not been created a similar application to the one made for this research. Since SwiftUI is only released half a year ago and Jetpack Compose is still in alpha stage, not enough time has passed for a company or an individual developer to create such a translator. Therefore we cannot compare our solution to an already existing implementation. We did however compare our solution to existing cross-platform mobile development tools in Chapter 2.

Chapter 4

Implementation

This research delivers a source to source translator from a subset of the language `Swift` (SwiftUI) to a subset of the language `Kotlin` (Jetpack Compose) and vice versa. The programs try to translate UIs from SwiftUI to Jetpack Compose and vice versa, although limitations are inevitable.
First of all, we had to decide the implementation approach. The decision came down to whether we preferred to use an existing library for parsing or to implement such a parser ourselves.

The choice we have made to create our program is to utilize language specific parsers. We have decided to develop the SwiftUI to Jetpack Compose translator in Swift and the Jetpack Compose to SwiftUI translator in Kotlin. The reasons for these choices are stated below.

- First of all, which is the main reason, Swift and Kotlin both offer a parser library for their respective frameworks. This is required, because to translate just the sub-languages, SwiftUI and Jetpack Compose, 'regular' Swift and Kotlin code can be written between the UI definitions. Using the already existing parser libraries means that we get the parsing part of our program for free, which would have been at least half the work. While we had working parsers created in `Haskell`, these could not parse the entire language Swift and we figured that it was outside the scope of our program to make sure that every possible input could be parsed correctly and would give reasonable and useful output.
- It would be superfluous to make the grammars for the languages `Swift` and `Kotlin` work within a self made parser in `Haskell` or using a parser tool like `parsec`[1], even though these grammars are available online.

[1] parsec, D. Leijen, P. Martini, A. Latter, 2006. `https://hackage.haskell.org/package/parsec`

All in all, we developed 2 distinct pieces of sofware: translaters both ways. `SwiftUISyntax`[2] translates a SwiftUI application to Jetpack Compose and `ComposeSyntax`[3] translates a Jetpack Compose application to SwiftUI. Both programs are command line tools and are available at our GitHub page and can be downloaded and used by developers.

To further clarify this choice, we listed all the tradeoffs below.

4.1 Existing Parser Libraries

The choice we opted for is to use libraries for Swift and Kotlin to translate from SwiftUI to Jetpack Compose and from Jetpack Compose to SwiftUI respectively.
`SwiftSyntax`[3] is such a library. It is created by Apple to create an *Abstract Syntax Tree* (AST) of Swift code. This way, we let these libraries handle the parsing for us. This means that we had to do less work, while using more robust software.
After parsing, we can walk through the generated AST using a visitor pattern. Section 4.3 explains how the AST is converted to the other framework.

4.2 Self Made Parser

Another option we considered is to implement a parser ourselves. This parser would then be used to generate an AST from the source code, after which the translation procedure would start. Within this option, we had two more options. We could write the parser for both SwiftUI and Jetpack Compose using combinators implemented in `Haskell`. Since we had experience with this, we tried to write a parser for SwiftUI. After we discovered that this would be too cumbersome, we considered our other option. Since parsers need to be created regularly by developers, tools are made for parsing. Examples of these tools are `parsec` for `Haskell` and `ANTLR`[4]. `parsec` abstracts away the monadic parser combinators by letting the user define the grammar of the to be parsed language. `ANTLR` works similarly, but is a more stand-alone application rather than a library. Since the complete Swift grammar is available online, this was a viable option.
Below we explain how such a parser and translator would work in `Haskell`, regardless of defining the parser ourselves or how `parsec` works under the hood.

Let us first take a look at a simple definition of a parser.

[2]SwiftUISyntax `https://github.com/ToineHulshof/SwiftUISyntax`
[3]ComposeSyntax `https://github.com/ToineHulshof/ComposeSyntax`
[4]ANTLR, T. Parr 1988. `https://www.antlr.org`

```
newtype Parser a = Parser
  { parse :: String -> Maybe (String, a)
  }
```

This parser has one function: `parse`. A parser with type `Parser a` gets as input the source code of a program, a string and tries to parse something with type `a`. This attempt may fail, therefore is the result of the function of type `Maybe b`. To be more precise, this function yields an object of type `a`, which is our AST and passes on the unparsed string. So part of the input string will be parsed, after which the remaining string is passed to the next parser.

The idea is to start with the fundamental (smallest) parser; a character parser. We combine multiple character parsers to create a parser that parses strings. These string parsers can then be combined to parse a specified line of code which adheres to the languages grammar. Ultimately we are combining parsers to parse the entire frameworks.
To create this functionality for our parser, we use the classes that introduce combinators; `Applicative` and `Alternative`. To use these classes, besides giving the instances of them, we als need to implement the `Functor` class. The instances are as follows.

```
instance Functor Parser where
  fmap f (Parser p) =
    Parser $ \input -> do
      (input', x) <- p input
      Just (input', f x)

instance Applicative Parser where
  pure x = Parser $ \input -> Just (input, x)
  (Parser p1) <*> (Parser p2) =
    Parser $ \input -> do
      (input', f) <- p1 input
      (input'', a) <- p2 input'
      Just (input'', f a)

instance Alternative Parser where
  empty = Parser $ \_ -> Nothing
  (Parser p1) <|> (Parser p2) =
      Parser $ \input -> p1 input <|> p2 input
```

Since combining these parsers until they are capable of parsing the entire SwiftUI and Jetpack Compose library would take significant effort, we decided to not go this route.

4.3 Code Explanation

This section explains how our implementations work and why certain decisions were made.
Even though some aspects occur in both programs, we decided to split the explanation in two parts to make it easier to understand.

4.3.1 SwiftUI → Jetpack Compose

First of all, we create a Jetpack Compose project using Gradle[5].
To parse Swift-code, the library `SwiftSyntax`[3], created by Apple is used. This library transforms a Swift source-code file to the AST. Then using the built in `SyntaxVisitors`, which use the visitor pattern, we walk though the tree.
Since the UI in SwiftUI is built from combined Views only, we translate whenever we encounter such a View. Thereafter, the statements of the view are split up. For every type that a statement can have, we call the function which translates that specific type.
To make the program not just skip over code that is not part of the UI of the application, we used the library SwiftKotlin[1] to translate non-UI code written in Swift to the Kotlin language. The alternative was to not translate these statements, but instead let the user of our program translate them manually. SwiftKotlin should help developers finalize their project even easier and quicker.
We list the steps taken by our program to translate the example code from Figure 3.1 to make the procedure clearer.

1. We encounter a `struct`. After we checked if it conforms to the `View` protocol, we start translating.
2. We translate each statement in the declarationlist of the `struct`.
3. First, we encounter a variable. Since this variable is marked with the annotation `@State`, we know this should be translated to a `state` variable in Jetpack Compose.
4. After the `count` variable is translated, we come across a property called `body`. We know this property defines the UI and should therefore be handled differently.
5. The outer view of the body is a `VStack`. This translates to a `Column`. Since we manually defined a list which contains the names of views that can have children (such as a `VStack`), we know that our translation does not stop here.

[5]Gradle, H. Dockter, A. Murdoch, S. Faber, P. Niederwieser, L. Daley, R. Gröschke, D. DeBoer, 2007. `https://gradle.org`

6. The `VStack` contains a `Button`. This button says 'Press me'. We set this text as the text of the button in Jetpack Compose. Since the code that should be executed after a button press is not part of the UI, it is translated using SwiftKotlin.

7. Next, we encounter a modifier. We first handle this modifier as opposed to the view the modifier belongs to, because in Jetpack Compose, a modifier is an argument to the view. So we need information about the modifiers before we translate the view. In this case, the color of the text should be made red.

8. Finally, we encounter the `Text` view. The text contains a string which uses string interpolation (`\(count)`). This should be taken into account when translating a string. Now we pass the modifier for changing the color of the text as an argument for the `Text` view.

The translated code can be found in Figure 3.2.
We used the tool `swift-ast`[6] to get an insight in how the AST looks like for an example program. This tool made us better understand the structure of the code we needed to process.

Synopsis

```
swiftuisyntax [-o path] path
```

The complete source code can be found in Appendix A or on `GitHub`[7]

4.3.2 Jetpack Compose → SwiftUI

This program works in essence in the same way as the SwiftUI to Jetpack Compose translator. It parses the `Kotlin` source-code using the library `Kastree`, after which we walk through the AST using the visitor pattern.
Whenever we encounter a statement that needs to be translated, we switch on the type of this statement. Now the function gets called that translates this statement type. Within such a statement, there could be multiple other statements. For each of these statements we again switch on the type and call the corresponding function. This is done until the statement we encounter does not contain statements itself and is therefore a leaf in the AST.
Sometimes a type behaves different in certain circumstances. For example the trailing lambda of a `Button` (the code within {}) contains statements that need to be executed when the button is clicked, but the trailing lambda of a `Column` contains statements that represents the children views of the `Column`. We take note of these circumstances and translate accordingly.

[6] swift-ast `https://swift-ast-explorer.kishikawakatsumi.com`
[7] SwiftUISyntax `https://github.com/ToineHulshof/SwiftUISyntax`

`Kastree` creates an AST which consists of nodes. A node contains an expression and a `tag`. It is up to the developer who uses `Kastree` to define what this `tag` should contain. In our case, we made the tag contain the string that represents the corresponding expression. This `tag` comes in handy when a simple expression could be parsed as numerous different types, which is outside our scope of translating UI. Whenever this happens, we append the `tag` to the translation.
Just like in the other program. We start with an empty list of `String`s. For every function annotated with `@Composable`, we translate that function. Afterwards we combine the translations and output them to a file.
Depending on different views and modifiers we encounter, the correct action is taken to translate that part of the AST.
Modifiers can be added to a view to modify it. We saw that in Jetpack Compose, a modifier is an argument of view, while in SwiftUI, a modifier is a function called on a view. We need to take this into account while translating.
We list the steps taken by our program to translate the example code from Figure 3.2 to make the procedure clearer.

1. We encounter a `fun`ction. After we confirmed it is annotated with `@Composable`, we start translating.

2. We translate each statement in the declarationlist of the `fun`ction.

3. First, we encounter a variable. Since this variable is marked with `+state`, we know this should be translated to a variable marked with `@State` in SwiftUI.

4. After the `count` variable is translated, we come across an expression. Since this is an expression and not a declaration or a statement, we know this property defines the UI of the `Counter` view and should therefore be handled differently.

5. The outer view is a `Column`. This translates to a `VStack`. The `Column` contains modifiers that tell the view it should be placed in the center of the screen. In SwiftUI, this is standard, but our program does not recognize this sequence of modifiers and therefore translates them. This does not have an effect on the output however. Since we manually defined a list which contains the names of views that can have children (such as a `Column`), we know that our translation does not stop here.

6. The `Column` contains a `Button`. This button says 'Press me'. We set this text as the text of the button in SwiftUI. The code that should be executed after a button press (defined in the parameter `onClick`) is not translated by our program. Instead, we get the `tag` of this piece of code, which just prints the original code (`count.value++`). It is up

to the user to manually post-process this piece of code. In order to make the user aware of the fact that post-processing is required, we include a comment in the translated code explaining that adjustments are probably needed.

7. Next, we encounter a `Text` view. The text contains a string which uses string interpolation (${count}). This should be taken into account when translating a string. We also see that the `Text` has a `style`. We predefined that a `style` corresponds with a modifier in SwiftUI. Therefore we translate it as such and it will become `.foregroundColor(.red)`.

The translated (and *post-processed*) code can be found in Figure 3.1.

Synopsis

```
composesyntax [-o path] path
```

The complete source code can be found in Appendix B or on `GitHub`[8].

[8] ComposeSyntax `https://github.com/ToineHulshof/ComposeSyntax`

Chapter 5

Results

In this chapter we have a look at two small, but real world apps that were translated by our programs. We compare the generated code to the original app and list the post-processing steps that were required to make the code compilable and if necessary, adjust the code to make the resulting app resemble the original app better.

5.1 Landmarks

The Landmarks application[1] is an application created by Apple to showcase their new framework SwiftUI. The application parses a local `JSON` file which contains information about landmarks around the world. These landmarks are nicely displayed in the UI. A user can navigate through these landmarks, add them to their favorites and is able to get more information by entering the detail-view where details about that landmark are displayed.

Example

To show the result of our translator, we present a piece of code from the original application and compare it to the generated code.
This example represents a Landmark Detailscreen, which can be seen in Figure 5.5. This example is chosen, because it contains all the characteristics of the program, including the elements that need post-processing.
In the code, circled numbers (e.g. ①) are used to indicate the positions where an adjustment is needed.
We detect the places where manual post-processing is required while walking through the AST, so to help the user of our program, we indicate with a comment where adjustments are required. This comment also gives an idea of what needs to be changed and why we cannot translate it directly.
These comments are replaced by circled numbers in the code in figure 5.2.

[1] Apple, SwiftUI Landmarks Tutorial, 2019. `https://developer.apple.com/tutorials/swiftui/tutorials`

```
struct LandmarkDetail: View {
    @EnvironmentObject var userData: UserData
    var landmark: Landmark

    var landmarkIndex: Int {
        userData.landmarks.firstIndex(where: { $0.id == landmark.id })!
    }

    var body: some View {
        VStack {
            MapView(coordinate: landmark.locationCoordinate)
                .edgesIgnoringSafeArea(.top)
                .frame(height: 300)

            CircleImage(imageName: landmark.imageName)
                .offset(x: 0, y: -130)
                .padding(.bottom, -130)

            VStack(alignment: .leading) {
                HStack {
                    Text(landmark.name)
                        .font(.title)

                    Button(action: {
                        self.userData.landmarks[self.landmarkIndex]
                            .isFavorite.toggle()
                    }) {
                        if self.userData.landmarks[self.landmarkIndex]
                            .isFavorite {
                            Image(systemName: "star.fill")
                                .foregroundColor(Color.yellow)
                        } else {
                            Image(systemName: "star")
                                .foregroundColor(Color.gray)
                        }
                    }
                }

                HStack(alignment: .top) {
                    Text(landmark.park)
                        .font(.subheadline)
                    Spacer()
                    Text(landmark.state)
                        .font(.subheadline)
                }
            }
            .padding()

            Spacer()
        }
    }
}
```

Figure 5.1: The original SwiftUI code for the Landmark Detailscreen (Figure 5.5a)

```
@Composable
fun LandmarkDetail(landmark: Landmark) {
    val landmarkIndex: Int {
        ①userData.landmarks.firstIndex(where: { $0.id == landmark.id })②!
    }

    Column(modifier = Modifier.fillMaxSize() + Modifier.wrapContentSize(Alignment.
    ↪ Center), arrangement = Arrangement.Center) {
        Box(modifier = Modifier.fillMaxWidth() + Modifier.preferredHeight(300.dp)) {
            MapView(landmark.coordinates)
        }
        Box(modifier = Modifier.fillMaxWidth() + Modifier.padding(bottom = ③(-130).dp)
    ↪ + Modifier.offset(x: 0.dp, y: ③(-130).dp), gravity = ContentGravity.Center) {
            CircleImage(landmark.imageName)
        }
        Column(Modifier.wrapContentSize(Alignment.CenterStart) + Modifier.padding(10.dp
    ↪ )) {
            Row {
                Text(landmark.name, style = TextStyle(fontSize = TextUnit.Em(4)))
                Clickable(onClick = {
                   ④self.landmarks[④self.landmarkIndex].isFavorite.②toggle()
                }) {
                    if (④self.userData.landmarks[④self.landmarkIndex].isFavorite) {
                        Icon(Icons.Rounded.⑤Star, tint = Color.Yellow)
                    } else {
                        Icon(Icons.Rounded.⑤Star, tint = Color.Gray)
                    }
                }
            }
            Row(modifier = Modifier.None) {
                Text(landmark.park, style = TextStyle(fontSize = TextUnit.Em(3)))
                Spacer(modifier = Modifier.weight(⑥1f, true))
                Text(landmark.state, style = TextStyle(fontSize = TextUnit.Em(3)))
            }
        }
        Spacer(modifier = Modifier.weight(⑥1f, true))
    }
}
```

Figure 5.2: The generated Jetpack Compose code for the Landmark Detailscreen (Figure 5.5b)

5.1.1 Adjustments

To make this piece of code compilable and correct i.e. resemble the original UI element, there were 6 adjustments made. The numbers in the following list correspond to the numbers in Figure 5.2. The item numbers are colored. Red means that a non-UI element was translated incorrectly by SwiftKotlin, which required an adjustment.
Blue means that the SwiftUI code couldn't feasibly be translated to Jetpack Compose, which implies an adjustment was needed.

1. SwiftUI uses a variable declaration with the property-wrapper `@EnvironmentObject` to define the state of the application that is accessible in every child-view of the parent where this variable is declared. The Jetpack Compose counterpart for this state is a class annotated with the `@Model` annotation. In Jetpack Compose, to access this model, a reference to the class is made. Since we do not know what the name for this reference is, we had to manually adjust it. In this case the lowercase variable name, used in SwiftUI, should be changed to a capitalized first letter.

2. The `firstIndex(where:)` function in Swift returns an optional, which means that whenever the result is used, one should unwrap this optional. In the original code, this variable is force-unwrapped with an exclamation mark (`firstIndex(where:)!`), which means that we assume that the result of `firstIndex(where:)` is not `nil`. Whenever the result was `nil` however, the code would crash.
   In Kotlin, such the function `firstIndex(where:)` also exists, but it does not return an optional (`firstIndexOrNull(where:)` does however). The unwrapping is already implicitly done, therefore the exclamation mark should be omitted.

3. Jetpack Compose does not have support for negative padding. To make sure the view is placed in its correct place, we made the `Box` (line 11) with the `CircleImage` (line 12) in it a child of the `Column` (line 14) defined below the `Box`.

4. Swift requires that every non-local variable in a closure should be referred to with `self`. Kotlin does not have this restriction, so we can omit these keywords.

5. SwiftUI and Jetpack Compose both have a default collection of icons. The icon names are translated in such a way that it has the highest chance of succeeding. For example `"star"` is translated to `Icon.Rounded.Star`, which coincidentally works in this case, but cannot be guaranteed in general.

6. Whenever a `Spacer` is used in SwiftUI, it pushes the surrounding views to the edges of the screen as far as possible. To achieve the same effect in Jetpack Compose, a weigth of 1 is given to the `Spacer`. This works whenever there is a single `Spacer` in a view, but it can distort the positions of some views whenever multiple `Spacer`s are present. This can be resolved by manually adjusting the weigths.

Furthermore, there are some adjustments that must be made before the translated application will compile. The following list will contain these adjustments in order from most important to least important.

- Platform specific APIs are not translated, because that would overcomplicate translation. This means that we had to manually translate a `MapView` which uses MapKit to a `MapView` that uses the Google Maps API.
- We tried to generate a new Kotlin project with Gradle, but the newest version did not contain an option to create a new project that uses Jetpack Compose. Therefore we manually created a new project (using Gradle in IntellIJ) and move the translated files to the correct location within that project.
- We manually listed the screens that can be navigated to. This is required, because navigation is not build into Jetpack Compose yet, so for now this was the only solution.
- To be able to access all resources from the resources folder in Android, access to the context is required. We achieved this by adding a `MyApplication` class to `androidmanifest.xml`. We predefined this class and is included in every translation.
- It is recommended to have auto-import turned on, because Kotlin requires an import for every element that is used. We do not keep track of all the required imports, but let a modern IDE (like IntellIJ) automatically figure out which import are required whenever the project is opened.
- To use the correct version of Jetpack Compose, the `build.gradle` file has been amended.
- At the top of every file, a `package` name was added.

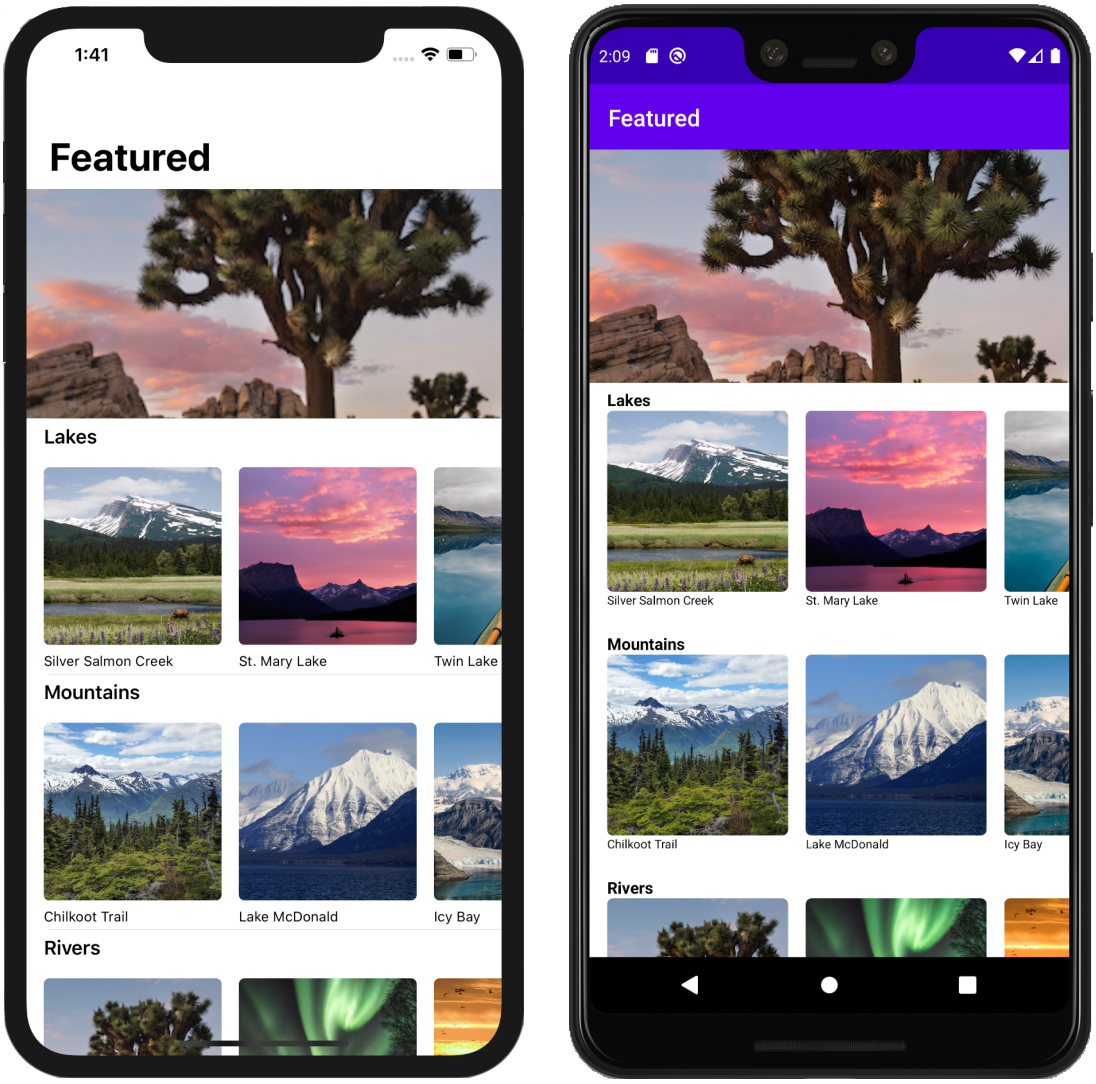
Although both lists are large, most of these issues only took a few seconds to a few minutes fix. In total the post-processing of this application took half an hour.

Figure 5.3 shows the generated code after the adjustments were made.

```
@Composable
fun LandmarkDetail(landmark: Landmark) {
    val landmarkIndex = UserData.landmarks.indexOfFirst { it.id == landmark.id }

    Column(modifier = Modifier.fillMaxSize() + Modifier.wrapContentSize(Alignment.
    ↪ Center), arrangement = Arrangement.Center) {
        Box(modifier = Modifier.fillMaxWidth() + Modifier.preferredHeight(300.dp)) {
            MapView(landmark.coordinates)
        }
        Column(Modifier.wrapContentSize(Alignment.CenterStart) + Modifier.padding(10.dp
    ↪ ) + Modifier.offset(0.dp, (-130).dp)) {
            Box(modifier = Modifier.fillMaxWidth() + Modifier.padding(10.dp), gravity =
    ↪  ContentGravity.Center) {
                CircleImage(landmark.imageName)
            }
            Row {
                Text(landmark.name, style = TextStyle(fontSize = TextUnit.Em(4)))
                Clickable(onClick = {
                    with(UserData) {
                        landmarks[landmarkIndex].isFavorite = !landmarks[landmarkIndex
    ↪ ].isFavorite
                        if (favorites.contains(landmark.id)) {
                            favorites.remove(landmark.id)
                        } else {
                            favorites.add(landmark.id)
                        }
                    }
                }) {
                    if (UserData.favorites.contains(landmark.id)) {
                        Icon(Icons.Rounded.Star, tint = Color.Yellow)
                    } else {
                        Icon(Icons.Rounded.Star, tint = Color.Gray)
                    }
                }
            }
            Row(modifier = Modifier.None) {
                Text(landmark.park, style = TextStyle(fontSize = TextUnit.Em(3)))
                Spacer(modifier = Modifier.weight(0.01f, true))
                Text(landmark.state, style = TextStyle(fontSize = TextUnit.Em(3)))
            }
        }
        Spacer(modifier = Modifier.weight(0.02f, true))
    }
}
```

Figure 5.3: The *post-processed* generated Jetpack Compose code for the Landmark Detailscreen (Figure 5.5b)

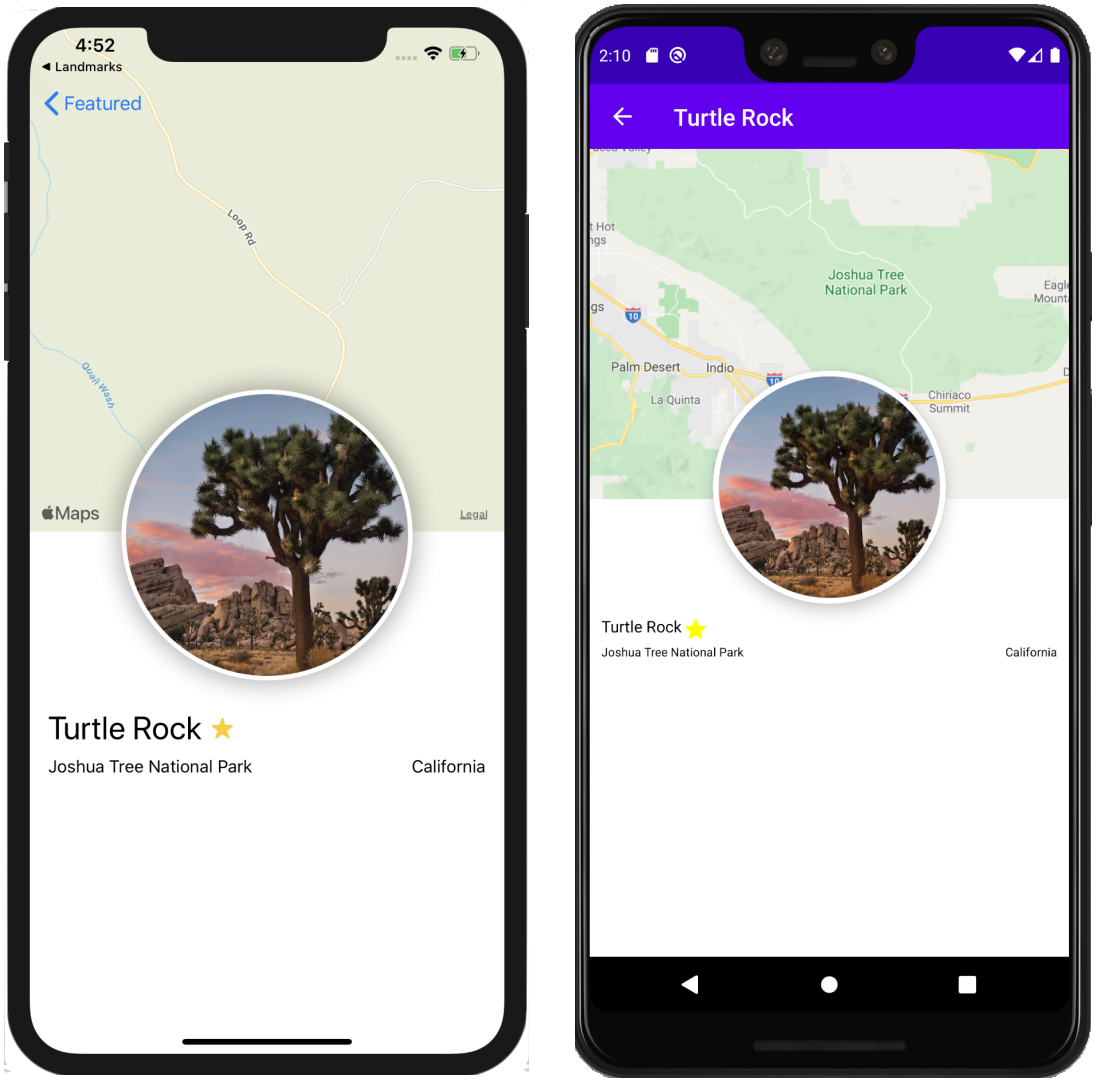
Below is a side-by-side comparison between the original SwiftUI (iOS) app and the *post-processed* generated Jetpack Compose (Android) app.



(a) SwiftUI (iOS)

(b) Jetpack Compose (Android)

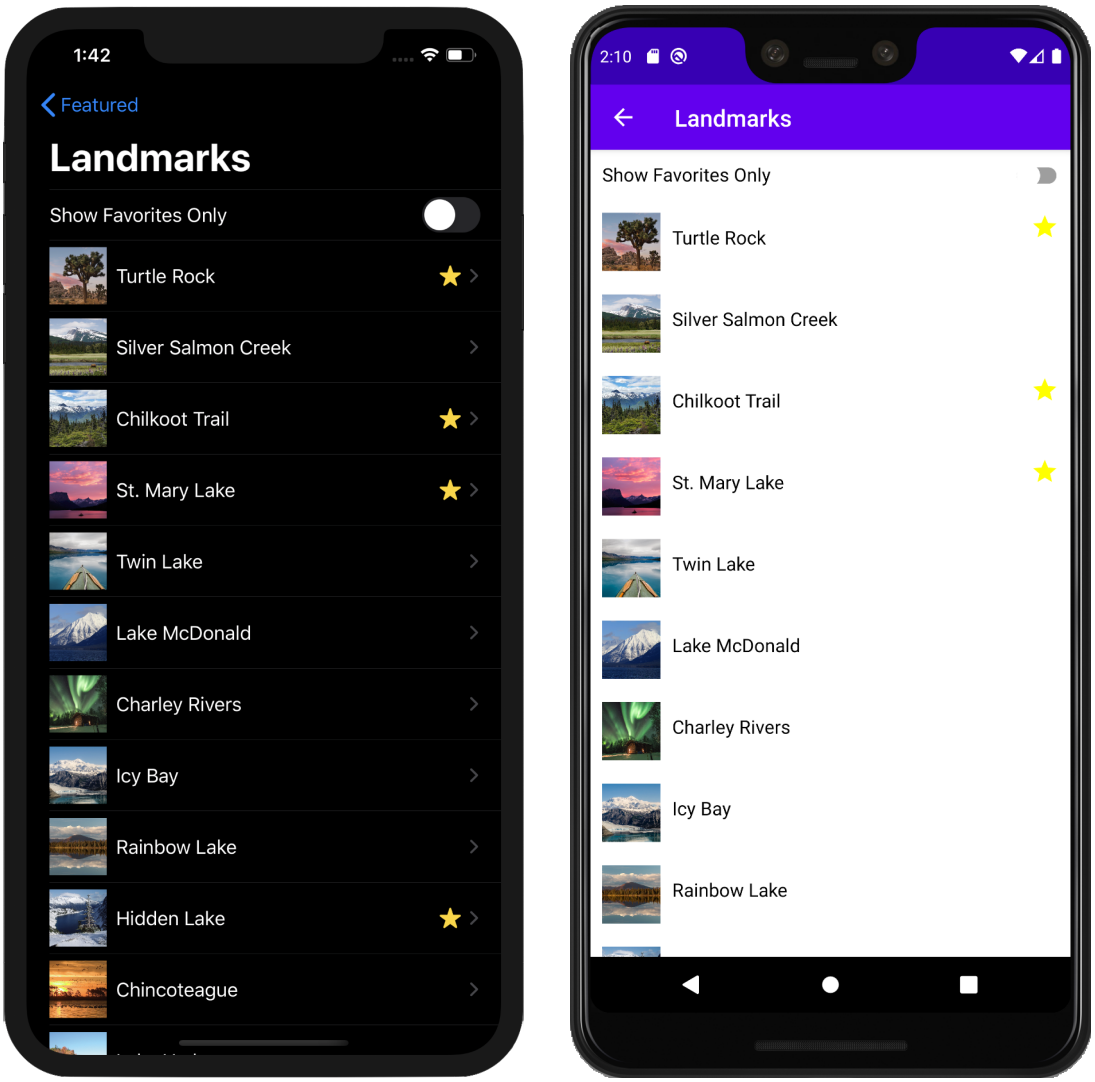
Figure 5.4: Landmarks Homescreen



(a) SwiftUI (iOS)

(b) Jetpack Compose (Android)

Figure 5.5: Landmark Detailscreen



(a) SwiftUI (iOS) (b) Jetpack Compose (Android)

Figure 5.6: Favorite Landmarks Overview

5.2 Jetnews

The Jetnews application is an application created by Google to showcase their new framework Jetpack Compose. This application is a prototype of a news application. The application unfortunately does not parse or fetch news articles from the internet, but instead stores articles in data classes, since `JSON` parsing and networking functionality is not yet build in.
The news articles are displayed in the UI on the homescreen in 4 different ways. A user can navigate through the articles and tap on one to read the entire article. A user can share or bookmark an article. There is also functionality build in to add article topics and interests to the favorites of users. This example demonstrates what the Jetpack Compose to SwiftUI translator accomplishes.

Example

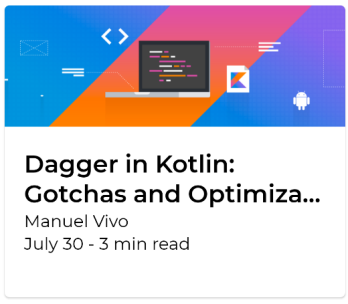
To show the result of our translator, we present a piece of code from the original application and compare it to the generated code.
This example represents a view for a `PostCard`, which can be seen in Figure 5.9. This example is chosen, because it contains all the strengths and weaknesses of the program, which we will elaborate upon.

```
@Composable
fun PostCardPopular(post: Post, modifier: Modifier = Modifier.None) {
    Card(modifier = modifier.preferredSize(280.dp, 240.dp), shape = RoundedCornerShape
    ↪ (4.dp)) {
        Clickable(
            modifier = Modifier.ripple(),
            onClick = { navigateTo(Screen.Article(post.id)) }
        ) {
            Column {
                val image = post.image ?: imageResource(R.drawable.placeholder_4_3)
                Image(image, Modifier.preferredHeight(100.dp).fillMaxSize())
                Column(modifier = Modifier.padding(16.dp)) {
                    val emphasisLevels = EmphasisAmbient.current
                    ProvideEmphasis(emphasisLevels.high) {
                        Text(
                            text = post.title,
                            style = MaterialTheme.typography.h6,
                            maxLines = 2,
                            overflow = TextOverflow.Ellipsis
                        )
                        Text(
                            text = post.metadata.author.name,
                            maxLines = 1,
                            overflow = TextOverflow.Ellipsis,
                            style = MaterialTheme.typography.body2
                        )
                    }
                    ProvideEmphasis(emphasisLevels.high) {
                        Text(
                            text = "${post.metadata.date} - " +
                                    "${post.metadata.readTimeMinutes} min read",
                            style = MaterialTheme.typography.body2
                        )
                    }
                }
            }
        }
    }
}
```

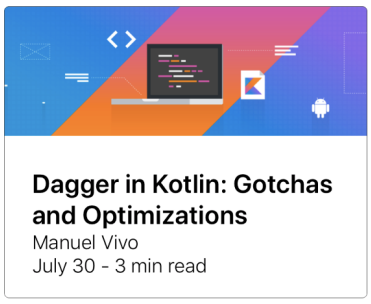
Figure 5.7: The original Jetpack Compose code for the '`PostCardPopular`' function (Figure 5.9a)

```
struct PostCardPopular: View {
  let post: Post

  var body: some View {
    Group {
      NavigationLink(destination: { ①// TODO: implement navigation -> navigateTo(Screen.
    ↪ Article(post.id)) }) {
        VStack(alignment: .leading) {
          Image(②image)
            .resizable()
            .frame(height: 100)
            .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .infinity)
            ③
          VStack(alignment: .leading) {
            Text(post.title)
              .bold()
              .font(.system(size: 20))
              .lineLimit(2)
            Text(post.metadata.author.name)
              .lineLimit(1)
              .font(.subheadline)
              .fontWeight(.light)
            Text("\(post.metadata.date) - \(post.metadata.readTimeMinutes) min read")
              .font(.subheadline)
              .fontWeight(.light)
          }
            .padding(16)
        }
      }
        .buttonStyle(PlainButtonStyle())
    }
      .overlay(RoundedRectangle(cornerRadius: 4).stroke(Color.init(white: 0.25),
    ↪ lineWidth: 1))
      .clipShape(RoundedCornerShape(4.dp)())
      .frame(width: 280, height: 240)
  }

}
```

Figure 5.8: The generated SwiftUI code for the '`PostCardPopular`' View (Figure 5.9b)



(a) Jetpack Compose (Android)



(b) SwiftUI (iOS)

Figure 5.9: '`PostCardPopular`'

5.2.1 Adjustments

To make the code in Figure 5.8 compilable and correct i.e. resemble the original UI element, there were 3 adjustments made. The numbers in the following list correspond with the numbers the figure. This time no colors are used, because we do not translate non UI elements whereas we used a library in the Landmarks example for these translations. Therefore no mistakes can be made in these non UI elements.

1. Since Jetpack Compose has not implemented functionality to navigate between screens as of May 2020, we cannot know to what screen should be navigated to whenever a certain button is clicked. We handled this limitation by adding a comment in the code which reminds the user that some adjustment is required. The user should write the identifier of the view that should be navigated to at the placeholder we provided for them.

2. Since Jetpack Compose references assets in a different way than SwiftUI does, images are handled slightly different as well. In the original Jetpack Compose code, an image is defined as a variable, where in the translated SwiftUI code, we made the manual adjustment that an image is a property of the `Post` structure. This means that we need to reference the image in a different way.

3. The way images are displayed in Jetpack compose also differs from SwiftUI. Since we cannot derive from the code how an image should be rendered in SwiftUI, we manually had to add the modifiers `aspectRatio(contentMode:)` and `clipped()` to make the image fit in the display.

Furthermore, there are some adjustments that must be made before the translated application will compile. The following list contains these adjustments in order from most important to least important.

- Platform specific APIs are not translated, so in order to use them, translations have to be done manually.

- The Jetpack Compose to SwiftUI translator does not generate an Xcode project. For now, a new project has to be manually created after which the translated files should be moved to within this project.

- We did not cover every possible type a declaration (a variable for example) can have. Therefore, some of these declarations need to be translated manually. The types we did not cover were also outside of this research, since we focussed on only the UI.

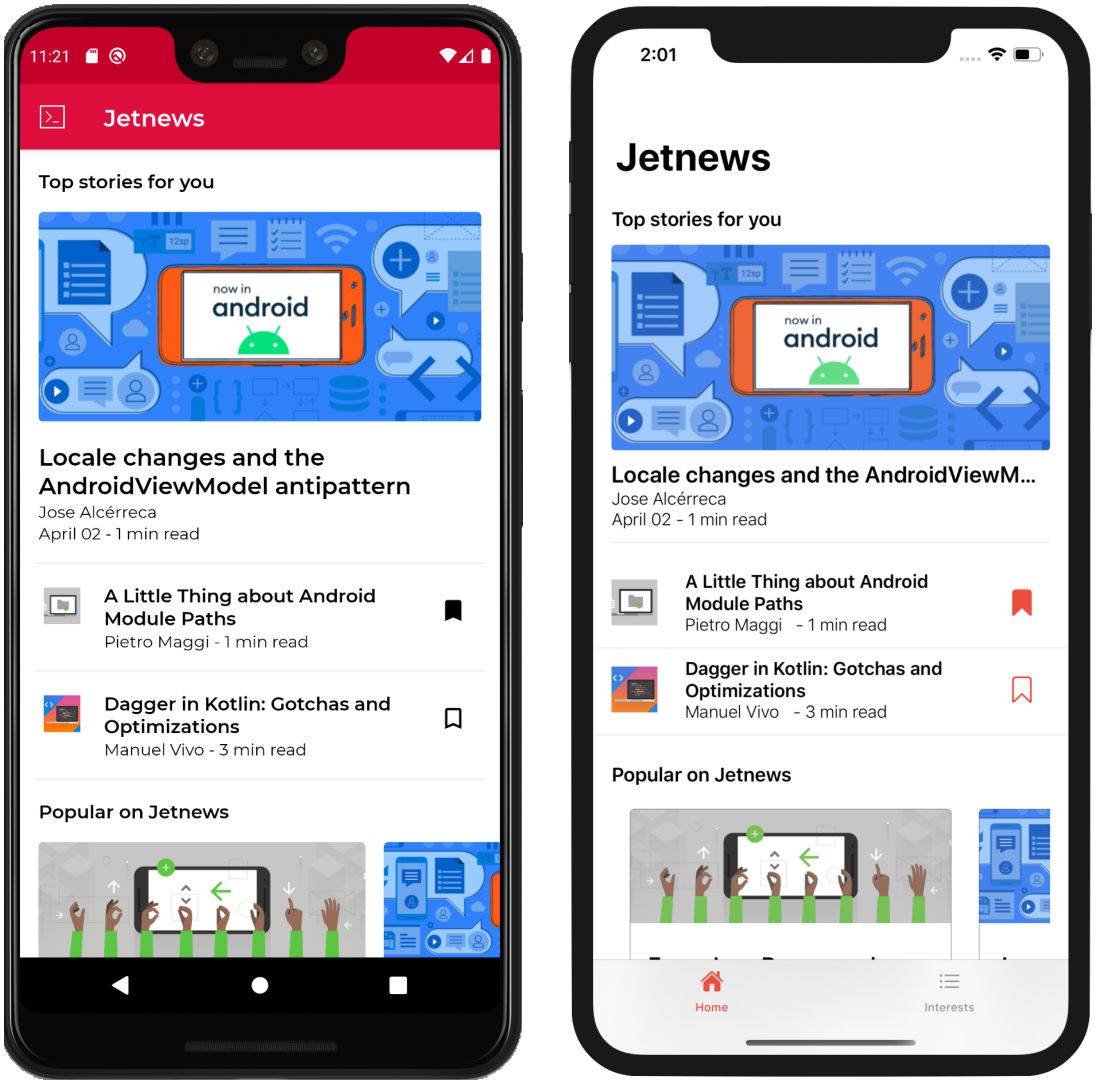
- iOS does not have an `AppDrawer`. To overcome this limitation, our program translated an `AppDrawer` to a `TabView`. However, since the definition of an element in an `AppDrawer` does not contain all the information required to translate it to a `TabView` item, some manual post-processing is required.
- Assets, including images, should be moved by hand to the assets folder.
- Not all icons in Android and iOS are the same and have the same names, therefore the used icons needed to be renamed to the correct icon.

Figure 5.10 shows the generated code with the aforementioned adjustments.

```swift
struct PostCardPopular: View {

    let post: Post

    var body: some View {
        VStack {
            NavigationLink(destination: ArticleScreen(post: post)) {
                VStack(alignment: .leading) {
                    Image(post.imageName)
                        .resizable()
                        .aspectRatio(contentMode: .fill)
                        .frame(height: 100)
                        .frame(minWidth: 0, maxWidth: .infinity)
                        .clipped()
                    VStack(alignment: .leading) {
                        Text(post.title)
                            .bold()
                            .font(.system(size: 20))
                        Text(post.metadata.author.name)
                            .font(.subheadline)
                            .fontWeight(.light)
                            .lineLimit(2)
                        Text("\(post.metadata.date) - \(post.metadata.readTimeMinutes) min read")
                            .lineLimit(1)
                            .font(.subheadline)
                            .fontWeight(.light)
                    }
                        .padding(16)
                }
            }
                .buttonStyle(PlainButtonStyle())
        }
            .overlay(RoundedRectangle(cornerRadius: 4).stroke(Color.init(white: 0.25), lineWidth: 1))
            .cornerRadius(4)
            .frame(width: 280, height: 240)
    }
}
```

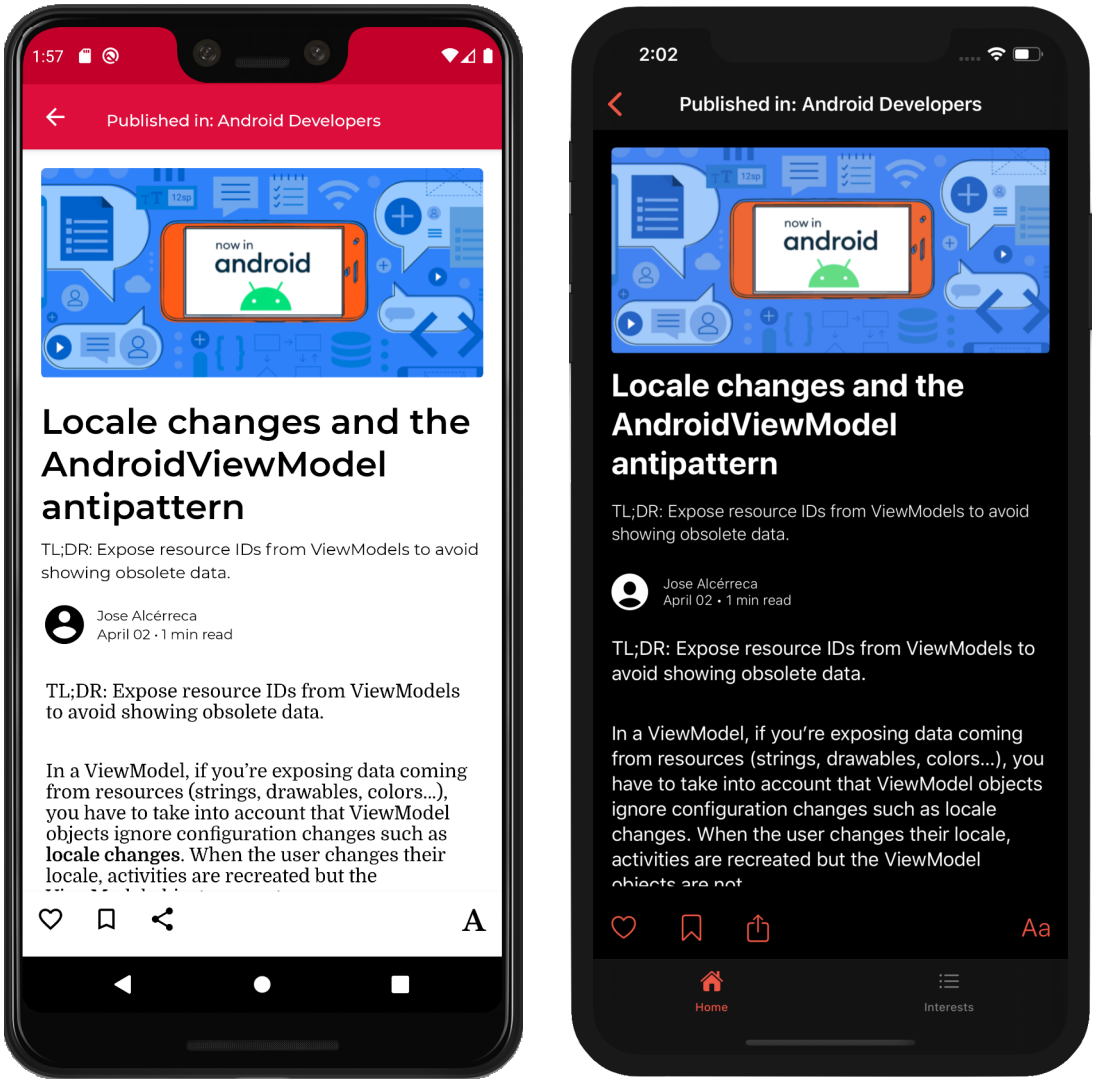
Figure 5.10: The *post-processed* generated SwiftUI code for the '`PostCardPopular`' View (Figure 5.9b)

Below is a side-by-side comparison between the original SwiftUI (iOS) app and the *post-processed* generated Jetpack Compose (Android) app.



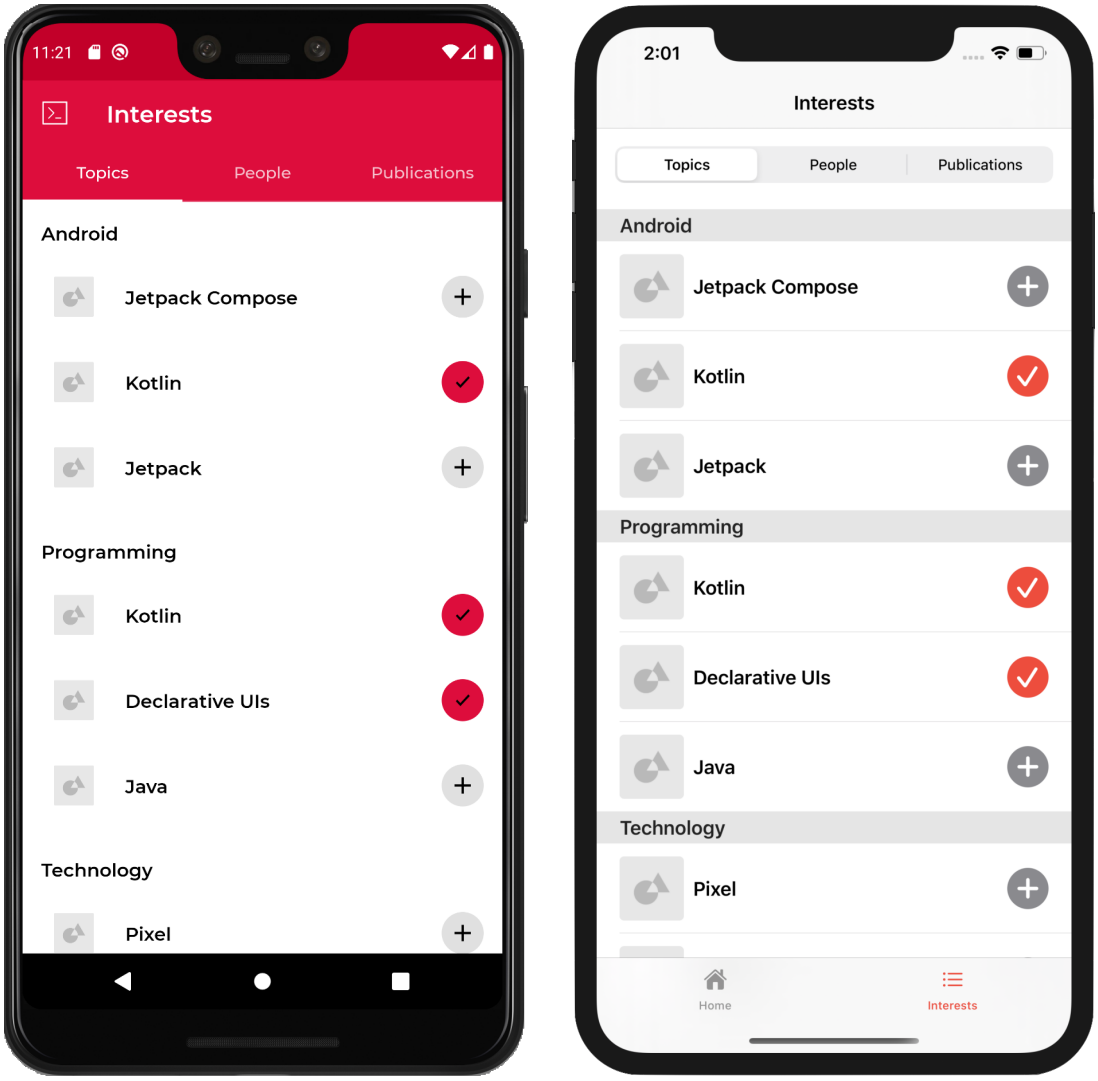
(a) Jetpack Compose (Android) (b) SwiftUI (iOS)

Figure 5.11: Jetnews Homescreen



(a) Jetpack Compose (Android) (b) SwiftUI (iOS)

Figure 5.12: News Article Detailview



(a) Jetpack Compose (Android) (b) SwiftUI (iOS)

Figure 5.13: Interests Overview

Since our programs' main focus is to translate UIs, we did not translate the code that is not part of the UI. For example, the `Button` on line 6 in figure 3.2, contains code that represents the button and code that is executed when the button is pressed. We translate the code that represents the button, because that is part of the UI. However, we do not translate the code within the `onClick` argument which is executed when the button is clicked, because this code is pure Kotlin code and is not part of the UI.
We do this, because there is no recently updated translation tool available that translates Kotlin to Swift. For the translation from SwiftUI to Jetpack Compose on the other hand, we used the library SwiftKotlin for these translations. For example, the code within the SwiftUI `Button` on line 7 in figure 3.1 will be translated.
Since we do not translate such code in the Jetpack Compose to SwiftUI translator, no mistakes in translating can happen. This explains why the list with adjustments is shorter than the list with adjustments for the Landmarks application.
Most of the adjustments and translations where rather easy and quick to apply. In total the post-processing of this application took approximately half an hour.

Chapter 6

Conclusions & Discussion

In this chapter we answer our research question

To what extend can a SwiftUI to Jetpack Compose translator (and vice versa) be developed and how does this implementation compare to the currently available cross-platform mobile development tools?

We do this by stating what we achieved with our programs, what its limitations are and how it compares with current cross-platform mobile development tools.

6.1 Proposed Solution

As can be seen from the other cross-platform options explained in Section 2, both SwiftUI and Jetpack Compose offer the best feature-set. The only *big* downside is that these frameworks are platform specific.
Our solution offers a tool for mobile app developers that can be used as a helper to translate their app from iOS to Android and vice versa. As has been stated before, our tool does not provide a full translation and manual editing of the generated source code is required.

Pros

- Our implementation allows developers to choose their preferred language and environment to develop in.
- Since our solution is native, hardware usage is better optimized which results in higher performance.
- When writing in the native language for the platform a developer is currently developing for, less code is required. This makes the code

more readable, easier to maintain and easier to enhance later on. A lot of boilerplate code used to define UI which is already standard in iOS or Android for example is left out. These characteristics combined enhances the code quality substantially.

Cons

- The biggest and only con of our tool is that is cannot translate and build an application without manual adjustment of the generated source code. Even though our programs try to translate as complete as possible, it remains infeasible to translate a platform specific API to its counterpart. However, when this translation is done manually, the performance increases since the mobile operating systems use their optimized, native APIs.

To conclude, our tool should be used whenever a developer would like to develop an app in a native environment and would also like to release their application for the other mobile operating system.
The developer must be willing to perform the required actions and adjustments to make the generated project run as expected on the other system.

As can be seen from the previous chapters, we managed to develop two applications that are able to make the conversion between iOS and Android applications easier.
Even though the frameworks SwiftUI and Jetpack Compose show similarities in declarative UI design and data flow, it is at this point still infeasible to create a complete program that does not need manual adjustment, because

- there are many differences between platform specific APIs
- there are some differences between SwiftUI and Jetpack Compose that make it ambiguous how to translate certain elements
- Jetpack Compose is still in pre-alpha stage, so the framework is bound to change in the future.

6.2 Limitations

The following list contains the two most note-worthy limitations of our program. These two drawbacks might withhold a developer from using our translator for production applications.

- Our program does not translate specific APIs. For example, `MapKit`, which we used to display Apple Maps was manually translated to support Google Maps support for Android.

- Our program only translates UI. To make a working application however, the UI is only part of the translation. What an application should perform after an UI interaction is defined using Swift for SwiftUI and Kotlin for Jetpack Compose. To be able to translate these languages into each other, we use 3^{rd} party libraries, however these libraries are not complete. This implies that translations done by these libraries should be manually post-processed, which might be a large effort, depending on the size of the code-base.

6.3 Comparison

According to the Stack Overflow Developer survey 2019[1], IDEs and workflows vastly differ from developer to developer. Hence this also applies to mobile application developers. We can however state that, if it were possible to translate an entire iOS or Android application to their mobile platform counterpart, it would be the ideal solution for all mobile application developers.
Such an program would however be infeasible to develop. However, we did not expect to develop a deterministic application that would be able to translate every application without manual adjustment.
To summarize, these are the conclusions from developing these applications and comparing them to existing cross-platform tools.

- It is infeasible to write an exhaustive source-to-source translator for Swift and Kotlin that generates compilable code. Until Apple supports Kotlin as a native iOS language, or Google supports Swift as a native Android language (in development), such a program will not likely exist.

- If one is interested in maintaining a single code base with no extra effort to make the application run on both platforms, other cross-platform alternatives like React Native or Flutter are preferred.

- If, however, one really prefers to develop in their familiar developing environment and seeks to make their application available to a broader audience, one of our programs can utilized. It requires minimal post-adjustment and the application runs native on both platforms.

[1]Stack Overflow, Stack overflow developer survey, 2019. Accessed at https://insights.stackoverflow.com/survey/2019 on 26-03-2020

Chapter 7

Future Work

The provided piece of software is not complete, as can be seen in the limitations.
In the research and software parts of this thesis, there are improvement possibilities for future work. We list some possible improvements that came up while conducting this research.

7.1 Research

During the research process, we thought of some research ideas one could conduct to further examine the possibilities of cross-platform programming for mobile app development.

1. A usability research. For example, give our programs to iOS and Android developers and perform qualitative analysis to research what they think about the provided program. Does this improve their workflow?, do they spend a lot of time post-processing the generated code? etc. Afterwards, the program could be improved with the obtained insights.

2. Is it actually faster to create a mobile application for iOS and Android by using this software as opposed to a cross-platform alternative?

7.2 Software

While programming this piece of software, the following enhancements came to mind:

1. The application could be made more complete. As of now, the core functionality is there, but there are some small improvements and adjustments that can be made. Section 6.2 lists some of the possible improvements.

2. The application could be updated to the newest Jetpack Compose and SwiftUI versions. Unfortunately, Jetpack Compose is still in early development, so an update is coming out almost every week. SwiftUI is in a further state, but is also subject to changes.

3. Since all the Kotlin code which is not part of the UI is not translated, a Kotlin to Swift source-to-source translator could be used. As of May 2020, such a library exist, but it only supports Swift version up to Swift 2.0, while currently the most recent Swift version is 5.2.

Bibliography

[1] Angelolloqui. SwiftKotlin. `https://github.com/angelolloqui/SwiftKotlin`, 2017. [Online; accessed 10-02-2020].

[2] Apple. Swift. `https://swift.org/documentation/`, 2014. [Online; accessed 22-01-2020].

[3] Apple. SwiftSyntax. `https://github.com/apple/swift-syntax`, 2018. [Online; accessed 26-03-2020].

[4] Apple. SwiftUI. `https://developer.apple.com/documentation/swiftui`, 2019. [Online; accessed 22-01-2020].

[5] B. Chown. Applying a single source of truth approach to the information needed for functional safety. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–5, 2018.

[6] Google. Flutter. `https://flutter.dev/docs`, 2018. [Online; accessed 22-01-2020].

[7] JetBrains. Kotlin. `https://kotlinlang.org/docs/reference/`, 2011. [Online; accessed 22-01-2020].

[8] JetBrains. Jetpack Compose. `https://developer.android.com/jetpack/compose`, 2019. [Online; accessed 22-01-2020].

[9] K. Shah, H. Sinha, and P. Mishra. Analysis of cross-platform mobile app development tools. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, pages 1–7, 2019.

[10] A. Syromiatnikov and D. Weyns. A journey through the land of model-view-design patterns. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 21–30, 2014.

[11] Y. Xia and M. Glinz. Rigorous ebnf-based definition for a graphic modeling language. In *Tenth Asia-Pacific Software Engineering Conference, 2003.*, pages 186–196, Dec 2003.

Appendix A

Swift Source Code

```swift
//
//  main.swift
//  SwiftUISyntax
//
//  Created by Toine Hulshof on 05/03/2020.
//  Copyright c 2020 Toine Hulshof. All rights reserved.
//

import SwiftSyntax
import Foundation

func shell(_ command: String) -> String {
    let task = Process()
    task.launchPath = "/bin/bash"
    task.arguments = ["--login", "-c", command]

    let pipe = Pipe()
    task.standardOutput = pipe
    task.launch()

    let data = pipe.fileHandleForReading.readDataToEndOfFile()
    let output: String = NSString(data: data, encoding: String.Encoding.utf8.rawValue)! as String

    return output
}

func translateFile(file: URL) -> String {
    do {
        let sourceFile = try SyntaxParser.parse(file)
        var visitor = SwiftUIVisitor()
        sourceFile.walk(&visitor)
        return visitor.translations.joined(separator: "\n\n")
    } catch {
        fatalError(error.localizedDescription)
    }
}

let input = CommandLine.arguments[1]
let inputURL = URL(fileURLWithPath: input)

if inputURL.hasDirectoryPath {
    let output = CommandLine.arguments[2]
    _ = shell("cd \(output)/app/src/main/java/com/translatedcompose && rm MainActivity.kt")

    FileManager.default.enumerator(at: inputURL, includingPropertiesForKeys: nil)?
        .compactMap({ $0 as? URL })
        .forEach { file in
        if file.hasDirectoryPath {
            let path = Array(file.pathComponents
                .suffix(file.pathComponents.count - inputURL.pathComponents.count))
                .map({ $0.replacingOccurrences(of: " ", with: "\\ ") })
            _ = shell("cd \(output)/app/src/main/java/com/toinehulshof/translatedcompose" +
                " && mkdir \(path.joined(separator: "/"))")
        }
        if file.pathExtension == "swift" {
            /// Translate file!
            var translation = ""
            switch file.lastPathComponent {
```

```
            case "AppDelegate.swift": return
            case "SceneDelegate.swift": translation = StructTranslator.sceneDelegateString
            default: translation = translateFile(file: file)
            }
            let path = Array(file.deletingPathExtension()
                .appendingPathExtension("kt").pathComponents
                .suffix(file.pathComponents.count - inputURL.pathComponents.count))
                .map({ $0.replacingOccurrences(of: " ", with: "\\ ")
                .replacingOccurrences(of: "SceneDelegate", with: "MainActivity") })
                _ = shell("cd \(output)/app/src/main/java/com/toinehulshof/translatedcompose" +
                    " && echo '\(translation)' > \(path.joined(separator: "/"))")
        } else {
            if file.hasDirectoryPath {

            } else {
                guard ["png", "jpg"].contains(file.pathExtension) else { return }
                _ = shell("cp \(file.path.replacingOccurrences(of: " ", with: "\\ "))" +
                    " \(output)/app/src/main/res/drawable-v24/\(file.lastPathComponent)")
            }
        }
    }
} else {
    /// Single File!
    let translation = translateFile(file: inputURL)
    do {
        try translation.write(to: inputURL.deletingPathExtension()
            .appendingPathExtension("kt"), atomically: false, encoding: .utf8)
    } catch {
        print(error.localizedDescription)
    }
}
```

```
//
//  StructTranslator.swift
//  SwiftUISyntax
//
//  Created by Toine Hulshof on 07/03/2020.
//  Copyright c 2020 Toine Hulshof. All rights reserved.
//

import Foundation
import SwiftSyntax
import Foundation

struct StructTranslator {

    let node: StructDeclSyntax
    private var isPreview: Bool {
        node.inheritanceClause?.description.contains("PreviewProvider") ?? false
    }
    private var isView: Bool {
        node.inheritanceClause?.description.contains("View") ?? false
    }

    func translate() -> String {
        var s = ""
        if !isView && !isPreview {
            KotlinTokenizer().translate(content: node.description).tokens?.forEach({ token in
                s += token.value
            })
            return s
        }
        if isPreview { s += "@Preview\n" }
        s += "@Composable\n"
        s += "fun \(node.identifier)"
        let (initializedMembers, uninitializedMembers, initializer) = splitMembers(node.members.
    ↪ members)
        if let initializer = initializer {
            s += initializer.parameters.description
        } else {
            s += "("
            s += uninitializedMembers.compactMap({ member in
                if let varDecl = member.decl as? VariableDeclSyntax {
                    return varDecl.bindings.description
                        .replacingOccurrences(of: "[", with: "List<")
                        .replacingOccurrences(of: "]", with: ">")
```

```
                }
                return nil
            }).joined(separator: ", ")
            s += ") "
        }
        s += "{" + newLine()
        initializedMembers.forEach { member in
            switch member.decl {
            case let varDecl as VariableDeclSyntax: s += tab() + translate(varDecl) + newLine()
            case let initDecl as InitializerDeclSyntax: print(initDecl.body?.description ?? "
↪ empty init")
            default: break
            }
        }
        s += "}"
        node.members.members.forEach { member in
            if let funcDecl = member.decl as? FunctionDeclSyntax {
                s += translateToKotlin(funcDecl) + newLine()
            }
        }
        return s
    }

    func splitMembers(_ members: MemberDeclListSyntax) ->
        ([MemberDeclListItemSyntax], [MemberDeclListItemSyntax], InitializerDeclSyntax?) {
        var initializedMembers = [MemberDeclListItemSyntax]()
        var uninitializedMembers = [MemberDeclListItemSyntax]()
        var initializer: InitializerDeclSyntax? = nil
        members.forEach { member in
            switch member.decl {
            case let varDecl as VariableDeclSyntax:
                var shouldTranslate = false
                varDecl.bindings.forEach { pattern in
                    if pattern.initializer != nil || ((pattern.accessor as? CodeBlockSyntax) !=
↪ nil) {
                        shouldTranslate = true
                    }
                }
                if let attributes = varDecl.attributes {
                    attributes.forEach { attribute in
                        if (attribute as? CustomAttributeSyntax) != nil {
                            shouldTranslate = true
                        }
                    }
                }
                shouldTranslate ? initializedMembers.append(member) : uninitializedMembers.append
↪ (member)
            case let initDecl as InitializerDeclSyntax: initializer = initDecl
            default: break
            }
        }
        return (initializedMembers, uninitializedMembers, initializer)
    }

    func translate(_ decl: DeclSyntax) -> String {
        switch decl {
        case let varDecl as VariableDeclSyntax: return translate(varDecl)
        case let funcDecl as FunctionDeclSyntax: return translateToKotlin(funcDecl)
        default: return "unsupported\n"
        }
    }

    func translate(_ decl: VariableDeclSyntax) -> String {
        var s = ""
        if let name = decl.bindings.firstToken, let view = getView(decl.bindings), name.text == "
↪ previews" {
            return "MaterialTheme {\n\t\t\(translate(view, 1))\n\t}"
        }
        if let name = decl.bindings.firstToken, let view = getView(decl.bindings), name.text == "
↪ body" {
            return translate(view, 2)
        }
        if !decl.withoutTrivia().description.starts(with: "@") {
            let translation = translateToKotlin(decl.withoutTrivia())
            return translation.trimmingCharacters(in: .whitespacesAndNewlines) + newLine()
        }
```

```
        switch decl.letOrVarKeyword.text {
        case "var": s += "val "
        case "let": s += "let "
        default: break
        }
        if let attribute = decl.attributes {
            s += decl.bindings.firstToken?.text ?? ""
            s += " = "
            switch attribute.description {
            case let str where str.contains("@State"): s += "+state{\(value(decl.bindings))}"
            case let str where str.contains("@ObservedObject"): s += "+observed{\(value(decl.
    ↪ bindings))}"
            case let str where str.contains("@Binding"): s += "+binding{\(value(decl.bindings))}"
            case let str where str.contains("@EnvironmentObject"):
                s += decl.bindings.firstToken?.text.capitalizingFirstLetter() ?? ""
                //s += "+environment{\(value(decl.bindings))}"
            case let str where str.contains("@FetchRequest"): s += "+fetch{\(value(decl.bindings)
    ↪ )}"
            default: break
            }
        } else {
            decl.bindings.forEach { binding in
                s += binding.withoutTrivia().description
            }
        }
        return s
    }

    func translate(_ codeBlock: CodeBlockSyntax, _ depth: Int) -> String {
        var s = ""
        codeBlock.statements.forEach { view in
            s += translateView(view, depth)
        }
        return s
    }

    func translateView(_ view: CodeBlockItemSyntax, _ depth: Int) -> String {
        var s = ""
        switch view.item {
        case let funcExpr as FunctionCallExprSyntax:
            switch funcExpr.calledExpression {
            case let idenExpr as IdentifierExprSyntax:
                s += translateViewBody(funcExpr, [], idenExpr, depth)
            case let memExpr as MemberAccessExprSyntax:
                s += translate(view: memExpr, [(memExpr.name.text, funcExpr.argumentList)], depth
    ↪ )
            default: break
            }
        case let memExpr as MemberAccessExprSyntax:
            if let base = memExpr.base {
                switch base {
                case let idenExpr as IdentifierExprSyntax:
                    switch idenExpr.identifier.text {
                    case "Color": s += "Color.\(memExpr.name.text.capitalizingFirstLetter())"
                    default: break
                    }
                default: break
                }
            }
        case let ifStmt as IfStmtSyntax:
            s += "if (" + translateToKotlin(ifStmt.conditions.withoutTrivia())
                .trimmingCharacters(in: .whitespacesAndNewlines) + ") {\n"
            ifStmt.body.statements.forEach { statement in
                s += tab(depth) + translateView(statement, depth + 1)
            }
            s += newLine() + tab(depth - 1) + "}"
            if let elseBody = ifStmt.elseBody as? CodeBlockSyntax {
                s += " else {\n"
                elseBody.statements.forEach { statement in
                    s += tab(depth) + translateView(statement, depth + 1)
                }
                s += newLine() + tab(depth - 1) + "}"
            }
//            print("if statement", ifStmt.body.)
        default: break
        }
```

```
188         return s
189     }
190 
191     func translate(view: MemberAccessExprSyntax, _ modifiers: [(String, Syntax)], _ depth: Int)
    ↪ -> String {
192         guard let base = view.base else { return "" }
193         var s = ""
194         switch base {
195         case let funcExpr as FunctionCallExprSyntax:
196             switch funcExpr.calledExpression {
197             case let mem as MemberAccessExprSyntax:
198                 s += translate(view: mem, modifiers + [(mem.name.text, funcExpr.argumentList)],
    ↪ depth)
199             case let iden as IdentifierExprSyntax:
200                 let (_, _, navigations, _, _, _) = translateModifiers(modifiers)
201                 if navigations.isEmpty {
202                     s += translateViewBody(funcExpr, modifiers, iden, depth)
203                 } else {
204                     s += "Scaffold(" + newLine() + tab(depth) + "topAppBar = {" +
205                         newLine() + tab(depth + 1) + "TopAppBar(" + newLine() + tab(depth + 2) +
206                         "title = { Text(text = \(navigations[0])) }" + newLine() + tab(depth + 1)
    ↪  + ")" +
207                         newLine() + tab(depth) + "}," + newLine() + tab(depth) + "bodyContent = {
    ↪ " +
208                         newLine() + tab(depth + 1) + translateViewBody(funcExpr, modifiers, iden,
    ↪  depth) +
209                         newLine() + tab(depth) + "}" + newLine() + tab(depth - 1) + ")"
210                 }
211             default: break
212             }
213         default: break
214         }
215         ///TODO: viewmodifiers toevoegen
216 //        s += newLine() + tab(depth) + "."
217 //        switch memExpr.name.text {
218 //        case "foregroundColor": s += "color"
219 //        default: s += "custom modifier"
220 //        }
221         return s
222     }
223 
224     func translateModifier(_ modifier: (String, Syntax)) -> String {
225         var s = ""
226         switch modifier.1 {
227         case let member as MemberAccessExprSyntax:
228             switch modifier.0 {
229             case "alignment": s += translateAlignment(member)
230             default: break
231             }
232         case let arguments as FunctionCallArgumentListSyntax:
233             switch modifier.0 {
234             case "padding": s += translatePadding(arguments)
235             case "frame": s += translateFrame(arguments)
236             case "shadow": s += translateShadow(arguments)
237             case "overlay": s += translateOverlay(arguments)
238             case "clipShape": s += translateClipShape(arguments)
239             case "font": s += translateFont(arguments)
240             case "foregroundColor": s += translateForegroundColor(arguments)
241             case "offset": s += translateOffset(arguments)
242             case "navigationBarTitle": s += translateNavigationBarTitle(arguments)
243             case "cornerRadius": s += translateCornerRadius(arguments)
244             case "scaledToFill": s += "ScaleFit.FillWidth"
245             case "scaledToFit": s += "ScaleFit.Fit"
246             default: s += "\(modifier.0) not implemented yet"
247             }
248         default: break
249         }
250         return s
251     }
252 
253     func translateNavigationBarTitle(_ arguments: FunctionCallArgumentListSyntax) -> String {
254         var title = ""
255         arguments.forEach { argument in
256             guard let expr = argument.expression as? FunctionCallExprSyntax else { return }
257             expr.argumentList.forEach { argument in
258                 guard let string = argument.expression as? StringLiteralExprSyntax else { return
```

```
    ↪ }
259                 title = translate(string)
260             }
261         }
262         return title
263     }
264
265     func translateCornerRadius(_ arguments: FunctionCallArgumentListSyntax) -> String {
266         var radius = "5"
267         arguments.forEach { argument in
268             guard let value = argument.expression as? IntegerLiteralExprSyntax else { return }
269             radius = value.withoutTrivia().description
270         }
271         return "Modifier.clip(RoundedCornerShape(\(radius).dp))"
272     }
273
274     func translateOffset(_ arguments: FunctionCallArgumentListSyntax) -> String {
275         var x = 0
276         var y = 0
277         arguments.forEach { argument in
278             guard let label = argument.label?.text,
279                 let value = argument.expression as? IntegerLiteralExprSyntax,
280                 let int = Int(value.withoutTrivia().description), int >= 0 else { return }
281             // Cannot translate negative padding, because App will crash
282             switch label {
283             case "x": x = int
284             case "y": y = int
285             default: break
286             }
287         }
288         return "Modifier.offset(x = \(x).dp, y = \(y).dp)"
289     }
290
291     func translateForegroundColor(_ arguments: FunctionCallArgumentListSyntax) -> String {
292         var color = "black"
293         arguments.forEach { argument in
294             guard let mem = argument.expression as? MemberAccessExprSyntax else { return }
295             switch mem.name.text {
296             case "primary": color = "black"
297             case "secondary": color = "gray"
298             default: color = mem.name.text
299             }
300         }
301         return "color = Color.\(color.capitalizingFirstLetter())"
302     }
303
304     func translateClipShape(_ arguments: FunctionCallArgumentListSyntax) -> String {
305         var shape = "Rectangle"
306         arguments.forEach { argument in
307             guard let expr = argument.expression as? FunctionCallExprSyntax,
308                 let iden = expr.calledExpression as? IdentifierExprSyntax else { return }
309             shape = iden.identifier.text
310         }
311         return "Modifier.clip(\(shape)Shape)"
312     }
313
314     func translateOverlay(_ arguments: FunctionCallArgumentListSyntax) -> String {
315         var color = "black"
316         var width = "5"
317         arguments.forEach { argument in
318             guard let shape = argument.expression as? FunctionCallExprSyntax else { return }
319             // Not full support for other kinds of overlays. Just borders for now
320             shape.argumentList.forEach { argument in
321                 if let colorMember = argument.expression as? MemberAccessExprSyntax {
322                     color = colorMember.name.text
323                 }
324                 if let label = argument.label?.text, label == "lineWidth",
325                     let value = argument.expression as? IntegerLiteralExprSyntax {
326                     width = value.withoutTrivia().description
327                 }
328             }
329         }
330         return "Border(\(width).dp, Color.\(color.capitalizingFirstLetter())"
331     }
332
333     func translateShadow(_ arguments: FunctionCallArgumentListSyntax) -> String {
```

```
        var radius = "10"
        arguments.forEach { argument in
            guard let label = argument.label?.text, label == "radius",
                let value = argument.expression as? IntegerLiteralExprSyntax else { return }
            radius = "\(value.withoutTrivia().description)"
        }
        return "Modifier.drawShadow(shape = CircleShape, elevation = \(radius).dp)"
    }

    func translateAlignment(_ member: MemberAccessExprSyntax) -> String {
        var s = "Modifier.wrapContentSize(Alignment."
        switch member.withoutTrivia().description {
        case ".top": s += "TopCenter"
        case ".bottom": s += "BottomCenter"
        case ".leading": s += "CenterStart"
        case ".trailing": s += "CenterEnd"
        default: s += "Center"
        }
        s += ")"
        return s
    }

    func translateFrame(_ arguments: FunctionCallArgumentListSyntax) -> String {
        var modifiers = [String]()
        arguments.forEach { argument in
            guard let label = argument.label?.text,
                let value = argument.expression as? IntegerLiteralExprSyntax else { return }
            var modifier = "Modifier."
            switch label {
            case "height": modifier += "preferredHeight(\(value.withoutTrivia().description).dp)"
            case "width": modifier += "preferredWidth(\(value.withoutTrivia().description).dp)"
            case "minWidth": modifier += "Modifier.fillMaxWidth()"
            case "minHeight": modifier += "Modifier.fillMaxHeight()"
            default: break
            }
            modifiers.append(modifier)
        }
        return modifiers.joined(separator: " + ")
    }

    func translatePadding(_ arguments: FunctionCallArgumentListSyntax) -> String {
        var s = "Modifier.padding("
        var edges = [String]() // Can also be implemented as a set, but then order is not
    ↪ preserved.
        var padding = "10.dp"
        arguments.forEach { argument in
            switch argument.expression {
            case let memExpr as MemberAccessExprSyntax: edges += getEdge(memExpr.name.text)
            case let array as ArrayExprSyntax:
                array.elements.forEach { element in
                    guard let memExpr = element.expression as? MemberAccessExprSyntax else {
    ↪ return }
                    edges += getEdge(memExpr.name.text)
                }
            case let intLit as IntegerLiteralExprSyntax: padding = intLit.withoutTrivia().
    ↪ description + ".dp"
            default: break
            }
        }
        s += edges.isEmpty ? padding : edges.map({ "\($0) = \(padding)" }).joined(separator: ", "
    ↪ )
        s += ")"
        return s
    }

    func getEdge(_ edge: String) -> [String] {
        switch edge {
        case "top": return ["top"]
        case "bottom": return ["bottom"]
        case "leading": return ["start"]
        case "trailing": return ["end"]
        case "horizontal": return ["start", "end"]
        case "vertical": return ["top", "bottom"]
        case "all": return ["top", "bottom", "start", "end"]
        default: return []
        }
```

```
406     }
407
408     func translateViewBody(_ viewBody: FunctionCallExprSyntax,
409                            _ modifiers: [(String, Syntax)],
410                            _ idenExpr: IdentifierExprSyntax,
411                            _ depth: Int) -> String {
412         var s = ""
413         ///TODO: zo veel mogelijk views toevoegen
414         switch idenExpr.identifier.text {
415         case "VStack": s += translateStack(.VStack, viewBody, modifiers, depth)
416         case "HStack": s += translateStack(.HStack, viewBody, modifiers, depth)
417         case "ZStack": s += translateStack(.ZStack, viewBody, modifiers, depth)
418         case "Text": s += translateText(viewBody, modifiers, depth)
419         case "Button": s += translateButton(viewBody, depth)
420         case "ForEach": s += translateForEach(viewBody, depth)
421         case "AngularGradient": s += "AngularGradient not implemented yet"
422         case "AnyView": s += "AnyView not implemented yet"
423         case "ButtonStyleConfiguration.Label": s += "ButtonStyleConfiguration.Label not
    ↪ implemented yet"
424         case "Color":
425             s += "Color."
426             print(viewBody)
427         case "DatePicker": s += "DatePicker not implemented yet"
428         case "Divider": s += "Divider()"
429         case "EditButton": s += "EditButton not implemented yet"
430         case "EmptyView": s += "EmptyView not implemented yet"
431         case "EquatableView": s += "EquatableView not implemented yet"
432         case "Form": s += "Form not implemented yet"
433         case "GeometryReader": s += "GeometryReader not implemented yet"
434         case "Group": s += "Group not implemented yet"
435         case "GroupBox": s += "GroupBox not implemented yet"
436         case "HSplitView": s += "HSplitView not implemented yet"
437         case "Image": s += translateImage(viewBody, modifiers, depth)
438         case "LinearGradient": s += "LinearGradient not implemented yet"
439         case "List":
440             guard let closureExpr = viewBody.trailingClosure else { return "" }
441             s += "VerticalScroller {\n"
442             s += tab(depth) + "Column {\n"
443             closureExpr.statements.forEach { view in
444                 s += tab(depth + 1) + translateView(view, depth + 2) + newLine()
445             }
446             s += tab(depth) + "}" + newLine()
447             s += tab(depth - 1) + "}"
448         case "MenuButton": s += "MenuButton not implemented yet"
449         case "ModifiedContent": s += "ModifiedContent not implemented yet"
450         case "NavigationLink":
451             var destination = ""
452             viewBody.argumentList.forEach { argument in
453                 guard let label = argument.label, label.text == "destination" else { return }
454                 if let view = argument.expression as? FunctionCallExprSyntax {
455                     let funcExpr = getLink(view)
456                     guard let identifier = funcExpr.calledExpression as? IdentifierExprSyntax
    ↪ else { return }
457                     destination += identifier.identifier.text
458                     destination += funcExpr.argumentList.count > 0 ? "(" + funcExpr.argumentList
459                         .map({ $0.expression.description })
460                         .joined(separator: ", ") + ")" : ""
461                 }
462             }
463             guard let closure = viewBody.trailingClosure else { return s }
464             var views = ""
465             closure.statements.forEach { statement in
466                 views += translateView(statement, depth + 1)
467             }
468             s += "Clickable(onClick = { navigateTo(Screen.\(destination)) }," +
469                 " modifier = Modifier.ripple() + Modifier.fillMaxWidth() + Modifier.padding(10.dp
    ↪ )) {" +
470                 newLine() + tab(depth) + views + newLine() + tab(depth - 1) + "}"
471
472         case "NavigationView":
473             guard let closureExpr = viewBody.trailingClosure else { return "" }
474             closureExpr.statements.forEach { statement in
475                 s += tab(depth) + translateView(statement, depth) + newLine()
476             }
477         case "Never": s += "Never not implemented yet"
478         case "Optional": s += "Optional not implemented yet"
```

```
        case "PasteButton": s += "PasteButton not implemented yet"
        case "Picker": s += "Picker not implemented yet"
        case "PrimitiveButtonStyleConfiguration.Label":
            s += "PrimitiveButtonStyleConfiguration.Label not implemented yet"
        case "RadialGradient": s += "RadialGradient not implemented yet"
        case "ScrollView":
            guard let closureExpr = viewBody.trailingClosure else { return "" }
            let (_, viewModifiers, _, _, _) = translateModifiers(modifiers)
            s += "HorizontalScroller"
            s += viewModifiers.isEmpty ? "" : "(modifier = " + viewModifiers.joined(separator: "
    ↪ + ") + ")"
            s += " {" + newLine()
            closureExpr.statements.forEach { statement in
                s += tab(depth) + translateView(statement, depth + 1) + newLine()
            }
            s += tab(depth - 1) + "}"
        case "Section":
            guard let closureExpr = viewBody.trailingClosure else { return "" }
            s += "Section {\n"
            closureExpr.statements.forEach { statement in
                s += tab(depth) + translateView(statement, depth + 1) + newLine()
            }
            s += tab(depth-1) + "}"
        case "SecureField": s += "SecureField not implemented yet"
        case "Slider": s += "Slider not implemented yet"
        case "Spacer":
            s += "Spacer(modifier = Modifier.weight(1f, true))"
        case "Stepper": s += "Stepper not implemented yet"
        case "SubscriptionView": s += "SubscriptionView not implemented yet"
        case "TabView": s += "TabView not implemented yet"
        case "TextField": s += "TextField not implemented yet"
        case "Toggle":
            var boolean = ""
            viewBody.argumentList.forEach { argument in
                guard let label = argument.label, label.text == "isOn" else { return }
                boolean = argument.expression.description.replacingOccurrences(of: "$", with: "")
            }
            s += "Switch(checked = \(boolean), onCheckedChange = { \(boolean) = it })"
            if let closure = viewBody.trailingClosure {
                s = "Row(modifier = Modifier.fillMaxWidth() + Modifier.padding(10.dp)" +
                    ", arrangement = Arrangement.SpaceBetween) {" + newLine() + tab(depth) +
                    closure.statements.map({ translateView($0, depth + 1) })
                        .joined(separator: newLine() + tab(depth)) +
                    newLine() + tab(depth) + s + newLine() + tab(depth - 1) + "}"
            }
        case "ToggleStyleConfiguration.Label": s += "ToggleStyleConfiguration.Label not
    ↪ implemented yet"
        case "TupleView": s += "TupleView not implemented yet"
        case "VSplitView": s += "VSplitView not implemented yet"
        default: s += translateCustom(viewBody, idenExpr, modifiers, depth)
        }
        s += ""
        return s
    }

    func getLink(_ funcExpr: FunctionCallExprSyntax) -> FunctionCallExprSyntax {
        switch funcExpr.calledExpression {
        case let member as MemberAccessExprSyntax:
            guard let base = member.base,
                let deepFunc = base as? FunctionCallExprSyntax else { return funcExpr }
            return getLink(deepFunc)
        case _ as IdentifierExprSyntax: return funcExpr
        default: return funcExpr
        }
    }

    func translateModifiers(_ viewModifiers: [(String, Syntax)]) ->
        ([String], [String], [String], [String], [String], [String]) {
        let reversedModifiers = viewModifiers.reversed()
        var borders = [(String, Syntax)]()
        var modifiers = [(String, Syntax)]()
        var navigations = [(String, Syntax)]()
        var scaleFits = [(String, Syntax)]()
        var styles = [(String, Syntax)]()
        var tints = [(String, Syntax)]()

```

```
        reversedModifiers.forEach { viewModifier in
            if ViewModifiers.borders.contains(viewModifier.0) { borders.append(viewModifier) }
            if ViewModifiers.modifiers.contains(viewModifier.0) { modifiers.append(viewModifier)
    ↪ }
            if ViewModifiers.navigations.contains(viewModifier.0) { navigations.append(
    ↪ viewModifier) }
            if ViewModifiers.scaleFits.contains(viewModifier.0) { scaleFits.append(viewModifier)
    ↪ }
            if ViewModifiers.styles.contains(viewModifier.0) { styles.append(viewModifier) }
            if ViewModifiers.tints.contains(viewModifier.0) { tints.append(viewModifier) }
        }

        return (borders.map({ translateModifier($0) }),
                modifiers.map({ translateModifier($0) }),
                navigations.map({ translateModifier($0) }),
                scaleFits.map({ translateModifier($0) }),
                styles.map({ translateModifier($0) }),
                tints.map({ translateModifier($0) }))
    }

    func translateImage(_ viewBody: FunctionCallExprSyntax,
                        _ modifiers: [(String, Syntax)],
                        _ depth: Int) -> String {
        var s = ""
        var imageName = ""
        var icon = false
        viewBody.argumentList.forEach { argument in
            if let name = (argument.expression as? StringLiteralExprSyntax)?
                .segments.withoutTrivia().description {
                icon = argument.label?.text == "systemName"
                imageName = name
            } else {
                imageName = argument.expression.description
            }

        }
        let (borders, translatedModifiers, _, scaleFits, _, tints) = translateModifiers(modifiers
    ↪ )
        if !icon {
            s += "Image(name = \(imageName)"
        } else {
            /// Icon
            s += "Icon(Icons.Rounded.\(imageName.components(separatedBy: ".")[0].
    ↪ capitalizingFirstLetter())"
            s += tints.isEmpty ? "" : ", \(tints.joined(separator: " + ").replacingOccurrences(of
    ↪ : "color", with: "tint"))"
            s += ")"
            return s
        }
        var newDepth = depth
        if !borders.isEmpty {
            newDepth += 1
            let shape = translatedModifiers
                .first(where: { $0.starts(with: "Modifier.clip") })?
                .components(separatedBy: "(")[1].components(separatedBy: ")")[0]
            s = "Box(\(shape != nil ? "shape = \(shape!), " : "")" +
                "border = \(borders.joined(separator: " + ")))" +
                "\(translatedModifiers.isEmpty ? "" : ", modifier = \(translatedModifiers.joined(
    ↪ separator: " + "))")" +
                ") {\n" + tab(depth) + s
        }
        if borders.isEmpty {
            s += translatedModifiers.isEmpty ? ""
                : ", modifier = \(translatedModifiers.joined(separator: " + "))"
        }
        s += scaleFits.isEmpty ? "" : ", scaleFit = \(scaleFits.joined(separator: " + "))"
        s += ")"
        if !borders.isEmpty {
            s += newLine() + tab(newDepth - 2) + "}"
        }
        return s
    }

    func translateForEach(_ viewBody: FunctionCallExprSyntax, _ depth: Int) -> String {
        guard let closure = viewBody.trailingClosure else { return "" }
        var s = ""
```

```
622         var isFirsArgument = true
623         viewBody.argumentList.forEach { argument in
624             if isFirsArgument {
625                 s += argument.expression.description
626                 isFirsArgument = false
627             }
628         }
629         s += ".forEach { \(closure.signature?.input?.description ?? "_ ")->\n"
630         closure.statements.forEach { statement in
631             s += tab(depth) + translateView(statement, depth + 1)
632         }
633         s += newLine() + tab(depth - 1) + "}"
634         return s
635     }
636 
637     func translateCustom(_ funcExpr: FunctionCallExprSyntax,
638                          _ idenExpr: IdentifierExprSyntax,
639                          _ modifiers: [(String, Syntax)],
640                          _ depth: Int) -> String {
641         var s = ""
642         s += idenExpr.identifier.text
643         s += "("
644         s += funcExpr.argumentList.map { argument in
645             argument.expression.description
646                 .replacingOccurrences(of: "$0", with: "it")
647                 .replacingOccurrences(of: "!", with: "!!")
648         }.joined(separator: ", ")
649         s += ")"
650         var (_, translatedModifiers, _, _, _, _) = translateModifiers(modifiers)
651         translatedModifiers.append("Modifier.fillMaxWidth()")
652 
653         s = "Box(modifier = \(translatedModifiers.joined(separator: " + "))" +
654             ", gravity = ContentGravity.Center) {" +
655             newLine() + tab(depth) + s + newLine() + tab(depth - 1) + "}"
656         return s
657     }
658 
659     func translateButton(_ funcExpr: FunctionCallExprSyntax, _ depth: Int) -> String {
660         var s = ""
661         guard let closure = funcExpr.trailingClosure else { return s }
662         if let actionExpr = funcExpr.argumentList
663             .first(where: { $0.label?.text == "action" }),
664             let action = actionExpr.expression as? ClosureExprSyntax {
665             s += "Clickable(onClick = {" + newLine()
666             s += translateSwift(action.statements, depth)
667             s += tab(depth - 1) + "}) {" + newLine() + tab(depth)
668             closure.statements.forEach { statement in
669                 s += translateView(statement, depth + 1)
670             }
671             s += newLine() + tab(depth - 1) + "}"
672         } else {
673             s += "Button(onClick = {" + newLine()
674             s += translateSwift(closure.statements, depth)
675             s += tab(depth - 1) + "}) {" + newLine() + tab(depth)
676             s += "Text(text = "
677             funcExpr.argumentList.forEach { argument in
678                 guard let string = argument.expression as? StringLiteralExprSyntax else { return
    ↪ }
679                 s += translate(string)
680             }
681             s += ")" + newLine() + tab(depth - 1) + "}"
682         }
683         return s
684     }
685 
686     func translateSwift(_ code: CodeBlockItemListSyntax, _ depth: Int) -> String {
687         var s = ""
688         code.forEach { statement in
689             if let seq = statement.item as? SequenceExprSyntax {
690                 s += tab(depth)
691                 seq.elements.forEach { expr in
692                     switch expr {
693                     case let member as MemberAccessExprSyntax: s += translateMember(member)
694                     case let biOp as BinaryOperatorExprSyntax: s += " \(biOp.withoutTrivia()) "
695                     case let intLit as IntegerLiteralExprSyntax: s += intLit.withoutTrivia().
    ↪ description
```

```
                    default : break
                    }
                }
                s += "\n"
            } else {
                s += tab(depth)
                KotlinTokenizer().translate(content: statement.description)
                    .tokens?.forEach({ token in
                    s += token.value
                })
            }
        }
        return s
    }

    func translateMember(_ memAccExpr: MemberAccessExprSyntax) -> String {
        return "\(memAccExpr.name.withoutTrivia()).value"
    }

    func translateText(_ funcExpr: FunctionCallExprSyntax,
                       _ modifiers: [(String, Syntax)],
                       _ depth: Int) -> String {
        var s = ""
        s += "Text(text = "
        funcExpr.argumentList.forEach { argument in
            if let string = argument.expression as? StringLiteralExprSyntax {
                s += translate(string)
            } else {
                s += argument.expression.description
            }
        }
        var (_, translatedModifiers, _, _, styles, _) = translateModifiers(modifiers)
        if !translatedModifiers.contains(where: { $0.contains("Modifier.padding") }) {
            translatedModifiers.append("Modifier.padding(5.dp)")
        }
        s += ", modifier = \(translatedModifiers.joined(separator: " + "))"
        s += styles.isEmpty ? "" : ", style = TextStyle(\(styles.joined(separator: ", ")))"
        s += ")"
        return s
    }

    func translateFont(_ arguments: FunctionCallArgumentListSyntax) -> String {
        var textSize: Double = 3
        var headline = false
        arguments.forEach { argument in
            guard let font = (argument.expression as? MemberAccessExprSyntax)?.name.text else {
    ↪ return }
            switch font {
            case "headline": headline = true
            case "caption": textSize = 2.5
            case "largeTitle": textSize = 5
            case "title": textSize = 4
            case "subheadline": textSize = 3
            case "callout": textSize = 2.5
            case "footnote": textSize = 2.5
            case "body": textSize = 2.5
            default: break
            }
        }
        return headline ? "fontWeight = FontWeight.Bold" : "fontSize = TextUnit.Em(\(textSize))"
    }

    func translate(_ string: StringLiteralExprSyntax) -> String {
        var s = "\""
        string.segments.forEach { segment in
            if let interPolatedString = segment as? ExpressionSegmentSyntax {
                interPolatedString.expressions.forEach { expr in
                    if expr.expression.description.contains("%") { return }
                    s += "${"
                    s += "\(expr.expression.description).value"
                    s += "}"
                }
            } else {
                s += segment.description
            }
        }
```

```
        s += "\""
        return s
    }

    func translateStack(_ stackType: StackType,
                        _ funcExpr: FunctionCallExprSyntax,
                        _ modifiers: [(String, Syntax)],
                        _ depth: Int) -> String {
        guard let closureExpr = funcExpr.trailingClosure else { return "" }
        var s = stackType.translation
        var allModifiers = modifiers
        funcExpr.argumentList.reversed().forEach { argument in
            guard let name = argument.label?.text else { return }
            allModifiers.append((name, argument.expression))
        }
        var (borders, translatedModifiers, _, _, _, _) = translateModifiers(allModifiers)
        translatedModifiers.append("Modifier.fillMaxSize()")
        if !translatedModifiers.contains(where: { $0.contains("wrapContentSize") }) {
            translatedModifiers
                .append("Modifier.wrapContentSize(Alignment.Center\(stackType == .HStack ? "Start
↪ " : ""))")
        }
        var newDepth = depth
        if !borders.isEmpty {
            newDepth += 1
            s = "Box(border = \(borders.joined(separator: " + "))" +
                "\(translatedModifiers.isEmpty ? "" : ", modifier = \(translatedModifiers.joined(
↪ separator: " + "))")" +
                ") {\n" + tab(depth) + s
        }
        s += "("
        if borders.isEmpty {
            s += translatedModifiers.isEmpty ? ""
                : "modifier = \(translatedModifiers.joined(separator: " + ")), "
        }
        s += "arrangement = Arrangement.Center)"
        s += " {\n"
        closureExpr.statements.forEach { view in
            s += tab(newDepth) + translateView(view, newDepth + 1) + newLine()
        }
        s += tab(newDepth - 1) + "}"
        if !borders.isEmpty {
            s += newLine() + tab(newDepth - 2) + "}"
        }
        return s
    }

    func getView(_ bindings: PatternBindingListSyntax) -> CodeBlockSyntax? {
        var codeSyntax: CodeBlockSyntax? = nil
        bindings.forEach { binding in
            codeSyntax = binding.accessor as? CodeBlockSyntax
        }
        return codeSyntax
    }

    func value(_ bindings: PatternBindingListSyntax) -> String {
        var s = ""
        bindings.forEach { pattern in
            s += pattern.initializer?.value.description ?? pattern.description
        }
        return s
    }

    func translateToKotlin(_ decl: Syntax) -> String {
        var s = "\n"
        let result = KotlinTokenizer().translate(content: decl.description)
        result.tokens?.forEach({ token in
            s += token.value
        })
        return s
    }

    func tab(_ count: Int = 1) -> String {
        String(repeating: "\t", count: count)
    }

```

```
    func newLine(_ count: Int = 1) -> String {
        String(repeating: "\n", count: count)
    }

}

extension String {
    func capitalizingFirstLetter() -> String {
        return prefix(1).capitalized + dropFirst()
    }

    mutating func capitalizeFirstLetter() {
        self = self.capitalizingFirstLetter()
    }
}
```

```
//
//  ImportTranslator.swift
//  SwiftUISyntax
//
//  Created by Toine Hulshof on 07/03/2020.
//  Copyright c 2020 Toine Hulshof. All rights reserved.
//

import Foundation
import SwiftSyntax

struct ImportTranslator {

    let node: ImportDeclSyntax

    func translate() -> String {
        switch node.path.description {
        case "SwiftUI":
            return """
            import androidx.compose.Composable
            import androidx.ui.animation.*
            import androidx.ui.core.*
            import androidx.ui.foundation.*
            import androidx.ui.framework.*
            import androidx.ui.layout.*
            import androidx.ui.material.*
            import androidx.ui.tooling.*
            """
        case "Combine":
            return "import androidx.compose.Model"
        default: return ""
        }
    }

}
```

```
//
//  VisitTokens.swift
//  SwiftUISyntax
//
//  Created by Toine Hulshof on 05/03/2020.
//  Copyright c 2020 Toine Hulshof. All rights reserved.
//

import Foundation
import SwiftSyntax

class SwiftUIVisitor: SyntaxVisitor {

    var translations = [String]()

    func visit(_ node: StructDeclSyntax) -> SyntaxVisitorContinueKind {
        translations.append(StructTranslator(node: node).translate())
        return .skipChildren
    }

    func visit(_ node: ImportDeclSyntax) -> SyntaxVisitorContinueKind {
        translations.append(ImportTranslator(node: node).translate())
        return .skipChildren
    }
```

```
25
26      func visit(_ node: ClassDeclSyntax) -> SyntaxVisitorContinueKind {
27          var s = ""
28          if node.inheritanceClause?.inheritedTypeCollection
29              .contains(where: {
30                  $0.withoutTrivia().typeName.description == "ObservableObject"
31              }) ?? false {
32              s += "@Model\n"
33              s += "object " + node.identifier.text + " {\n"
34              node.members.members.forEach { member in
35                  guard let varDecl = member.decl as? VariableDeclSyntax else { return }
36                  s += "\t\(varDecl.letOrVarKeyword.withoutTrivia()) " +
37                       "\(varDecl.bindings.withoutTrivia())\n"
38              }
39              s += "}"
40          } else {
41              KotlinTokenizer().translate(content: node.description).tokens?.forEach({ token in
42                  s += token.value
43              })
44          }
45          translations.append(s)
46          return .skipChildren
47      }
48
49      func visit(_ node: FunctionDeclSyntax) -> SyntaxVisitorContinueKind {
50          if node.identifier.text == "load" {
51              let s = """
52              fun getJsonDataFromAsset(context: Context, fileName: String): String? {
53                  val jsonString: String
54                  try {
55                      jsonString = context.assets.open(fileName).bufferedReader().use {
56                          it.readText()
57                      }
58                  } catch (ioException: IOException) {
59                      ioException.printStackTrace()
60                      return null
61                  }
62                  return jsonString
63              }
64              """
65              translations.append(s)
66          }
67          return .skipChildren
68      }
69
70      func visit(_ node: VariableDeclSyntax) -> SyntaxVisitorContinueKind {
71          node.bindings.forEach { binding in
72              guard let funcExpr = binding.initializer?.value as? FunctionCallExprSyntax,
73                  let type = binding.typeAnnotation else { return }
74              if funcExpr.calledExpression.description == "load" {
75                  let typeDescription = type.type.description
76                      .replacingOccurrences(of: "[", with: "Array<")
77                      .replacingOccurrences(of: "]", with: ">")
78                      .trimmingCharacters(in: .whitespaces)
79                  translations.append("val \(binding.pattern)" +
80                      "= Gson().fromJson(getJsonDataFromAsset(MyApplication.appContext!!," +
81                      "\(funcExpr.argumentList.description), \(typeDescription))::class.java)" +
82                      "\(typeDescription.contains("Array") ? ".asList()" : "")")
83              }
84          }
85          return .skipChildren
86      }
87
88      func visit(_ node: EnumDeclSyntax) -> SyntaxVisitorContinueKind {
89          var s = ""
90          KotlinTokenizer().translate(content: node.description).tokens?.forEach({ token in
91              s += token.value
92          })
93          translations.append(s)
94          return .skipChildren
95      }
96
97  }
```

```
1  //
2  //  ViewTypes.swift
```

```
//  SwiftUISyntax
//
//  Created by Toine Hulshof on 08/03/2020.
//  Copyright c 2020 Toine Hulshof. All rights reserved.
//

import Foundation

enum StackType {
    case HStack, VStack, ZStack
    var translation: String {
        switch self {
        case .HStack: return "Row"
        case .VStack: return "Column"
        case .ZStack: return "Stack"
        }
    }
}

struct ViewModifiers {
    static let borders = ["overlay"]
    static let modifiers = ["alignment", "padding", "frame", "shadow", "clipShape", "offset", "
    ↪ cornerRadius"]
    static let navigations = ["navigationBarTitle"]
    static let scaleFits = ["scaledToFill", "scaledToFit"]
    static let styles = ["font", "foregroundColor"]
    static let tints = ["foregroundColor"]
}
```

```
//
//  MainActivityText.swift
//  SwiftUISyntax
//
//  Created by Toine Hulshof on 15/04/2020.
//  Copyright c 2020 Toine Hulshof. All rights reserved.
//

import Foundation

extension StructTranslator {
    // This string is included in every project
    static let sceneDelegateString = """
    package //Change this to your package name

    import android.app.Application
    import android.content.Context
    import android.os.Bundle
    import androidx.appcompat.app.AppCompatActivity
    import androidx.compose.Composable
    import androidx.compose.Model
    import androidx.ui.animation.Crossfade
    import androidx.ui.core.ContextAmbient
    import androidx.ui.core.Modifier
    import androidx.ui.core.setContent
    import androidx.ui.foundation.Icon
    import androidx.ui.foundation.Text
    import androidx.ui.graphics.ScaleFit
    import androidx.ui.material.*
    import androidx.ui.material.icons.Icons
    import androidx.ui.material.icons.filled.ArrowBack
    import androidx.ui.res.imageResource

    class MainActivity : AppCompatActivity() {
        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContent {
                MaterialTheme {
                    Crossfade(Navigation.currentScreen) { screen ->
                        when (screen) {
                            // This is the main screen defined in SceneDelegate.swift
                            // (default is ContentView())
                            is Screen.Home ->
                                CategoryHome()
                            // Add custom screens
                            // is Screen.LandmarkList ->
                                NavigationLink(title = "Landmarks") { LandmarkList() }
```

```
                            // is Screen.Landmark ->
                                NavigationLink(title = screen.landmark.name) {
                                    LandmarkDetail(landmark = screen.landmark)
                                }
                        }
                    }
                }
            }
        }

        // add
        // <application
        //         android:name=".MyApplication"
        // in AndroidManifest.xml
        class MyApplication : Application() {
            override fun onCreate() {
                super.onCreate()
                context = applicationContext
            }

            companion object {
                private var context: Context? = null
                val appContext: Context?
                    get() = context
            }
        }

        sealed class Screen {
            object Home : Screen()
            // Add customs screens
            // object LandmarkList : Screen()
            // data class Landmark(val landmark: Landmark) : Screen()
        }

        @Model
        object Navigation {
            var currentScreen: Screen = Screen.Home
        }

        fun navigateTo(destination: Screen) {
            Navigation.currentScreen = destination
        }

        @Composable
        fun NavigationLink(title: String = "", destination: @Composable() () -> Unit) {
            Scaffold(
                topAppBar = {
                    TopAppBar(
                        title = {
                            Text(title)
                        },
                        navigationIcon = {
                            IconButton(onClick = { navigateTo(Screen.Home) }) {
                                Icon(Icons.Filled.ArrowBack)
                            }
                        }
                    )
                },
                bodyContent = {
                    destination()
                }
            )
        }

        @Composable
        fun Image(name: String, modifier: Modifier = Modifier.None, scaleFit: ScaleFit = ScaleFit.Fit
        ↪ ) {
            androidx.ui.foundation.Image(
                imageResource(
                    ContextAmbient.current.resources.getIdentifier(
                        name,
                        "drawable",
                        ContextAmbient.current.packageName
                    )
                ),
```

```
123             modifier = modifier,
124             scaleFit = scaleFit
125         )
126     }
127     """
128 }
```

Appendix B

Kotlin Source Code

```kotlin
import kastree.ast.Node
import kastree.ast.Visitor
import kastree.ast.psi.Converter
import kastree.ast.psi.Parser
import org.jetbrains.kotlin.com.intellij.psi.PsiElement
import org.jetbrains.kotlin.utils.fileUtils.withReplacedExtensionOrNull
import java.io.File

val ignoredParameters = listOf("ScaffoldState", "Modifier")
val ignoredArguments = ignoredParameters.map { it.decapitalize() } + listOf("checked", "overflow", "items", "value", "onCloseRequest", "confirmButton")
val ignoredViews = listOf("MaterialTheme", "Surface", "ProvideEmphasis", "ThemedPreview")
val ignoredParametersForViews = listOf("Scaffold")
val ignoredModifiers = listOf("ripple", "gravity")
val modifierArguments = listOf("style", "color", "scaleFit", "shape", "maxLines", "onValueChange", "backgroundColor")
val viewsWithNormalChildren = listOf("MaterialTheme", "Crossfade", "Surface", "VerticalScroller", "HorizontalScroller", "Column", "Row", "Stack", "ProvideEmphasis", "Clickable", "forEach", "ThemedPreview", "IconToggleButton", "Card", "Box", "TabRow", "Toggleable", "IconButton")

fun main(args: Array<String>) {
    if (args.isEmpty()) {
        println("Please enter a file or directory path")
        return
    }
    val file = File(args[0])

    file.walk().forEach {
        if (it.extension == "kt") {
            val translation = translateFile(it)
            println(translation)
            val outputFile = File(it.path.replace("jetnews", "swiftui"))
                .withReplacedExtensionOrNull("kt", "swift") ?: return@forEach
            outputFile.parentFile.mkdirs()
            if (outputFile.createNewFile()) {
                outputFile.writeText(translation)
                println("Translated ${it.path}")
                return@forEach
            }
            println("Failed to translate ${it.path}")
        }
    }
}

fun translateFile(file: File): String {
    val text = file.readText()
    val code = parseCodeWithText(text)
    val translations = mutableListOf<String>()
    Visitor.visit(code) { v, _ ->
        when (v) {
            is Node.Decl.Func -> {
                val annotations = v.mods
                    .mapNotNull { (it as? Node.Modifier.AnnotationSet)?.anns
                    ?.flatMap { it.names } }
                    .flatten()
                if (annotations.contains("Composable")) {
                    translations.add(translateComposable(v, annotations.contains("Preview"), 1))
```

```
                } else {
                    val tag = v.tag as? String ?: return@visit
                    translations.add(tag)
                }
            }
            is Node.Decl.Structured -> {
                val annotations = v.mods
                    .mapNotNull { (it as? Node.Modifier.AnnotationSet)?.anns
                    ?.flatMap { it.names } }
                    .flatten()
                if (annotations.contains("Model")) {
                    translations.add(translateModel(v))
                }
            }
        }
    }
    return translations.joinToString("\n\n")
}

fun parseCodeWithText(code: String) = Parser(object : Converter() {
    override fun onNode(node: Node, elem: PsiElement) {
        node.tag = elem.text
            .replace(" =", ":")
            .replace("?:", "??")
    }
}).parseFile(code)

fun translateModel(model: Node.Decl.Structured): String {
    var s = "class ${model.name}: ObservableObject {\n"
    s += model.members
        .filterIsInstance<Node.Decl.Property>()
        .filter { it.readOnly }
        .joinToString("\n") {
        var property = "\t"
        property += "@Published var " + it.vars.firstOrNull()?.name + " = "
        when (val expr = it.expr) {
            is Node.Expr.Call -> {
                (expr.expr as? Node.Expr.Name)?.let {
                    if (it.name == "ModelList")  {
                        property += "[${expr.typeArgs.firstOrNull()?.ref?.tag}]()"
                    }
                }
            }
        }
        return@joinToString property
    }
    s += "\n}"
    return s
}

fun translateComposable(func: Node.Decl.Func, isPreview: Boolean, depth: Int): String {
    var s = ""
    val parameters = func.params.
        filter {
            !ignoredParameters.
                contains((it.type?.ref as? Node.TypeRef.Simple)?.pieces
                ?.map { it.name }
                ?.firstOrNull())
        }
    val statements = (func.body as? Node.Decl.Func.Body.Block)?.block?.stmts ?: return s
    val declarations = statements.filterIsInstance<Node.Stmt.Decl>()
    val expressions = statements.filterIsInstance<Node.Stmt.Expr>()

    s += "struct ${func.name}: " + (if (isPreview) "PreviewProvider" else "View") + " {\n"
    s += if (parameters.isNotEmpty()) parameters
        .joinToString("\n", postfix = "\n\n") { tab(depth) + translateProperty(it)
            .replace("Boolean", "Bool")
            .replace("Unit", "Void") } else ""
    s += if (declarations.isNotEmpty()) declarations
        .joinToString("\n", postfix = "\n\n") {
            tab(depth) + translateDeclaration(it)
        } else ""
    s += tab(depth) + (if (isPreview) "static " else "") +
        "var ${if (isPreview) "previews" else "body"}" +
        ": some View {\n"
    s += if (expressions.size > 1)
```

```
        tab(depth + 1) + "VStack(alignment: .leading) {\n" else ""
    s += if (expressions.isNotEmpty()) expressions
        .joinToString("\n") {
            translateBody(
                it.expr,
                null,
                expressions.size,
                depth + if (expressions.size > 1) 2 else 1
            )
        } else ""
    s += if (expressions.size > 1) "\n" + tab(depth + 1) + "}" else ""
    s += "\n" + tab(depth) + "}" + if (parameters.size + declarations.size != 0) "\n" else ""
    s += "\n" + tab(depth - 1) + "}"
    return s
}

fun translateBody(expr: Node.Expr, parent: String?, childrenCount: Int, depth: Int): String {
    var isForEach = false
    when (expr) {
        is Node.Expr.When -> {
            if (parent == "Surface") {
                return expr.entries
                    .mapNotNull { it.body as? Node.Expr.Call }
                    .joinToString("\n") { translateTabItem(it, depth + 1) }
            }
            return expr.entries
                .joinToString("\n" + tab(depth), tab(depth)) {
                    "if ${expr.expr?.tag} == ${it.conds.firstOrNull()?.tag}" +
                        " { ${translateBody(it.body, parent, 1, 0)} }"
                }
        }
        is Node.Expr.If -> {
            var s = ""
            val stmts = (expr.body as? Node.Expr.Brace)?.block?.stmts ?: return s
            s += tab(depth) + "if ${expr.expr.tag} {\n" + stmts
                .mapNotNull { it as? Node.Stmt.Expr }
                .joinToString("\n") {
                    translateBody(it.expr, parent, stmts.size, depth + 1)
                } + "\n" + tab(depth) + "}"
            expr.elseBody?.let {
                val elseStmts = (it as? Node.Expr.Brace)?.block?.stmts ?: return s
                s += " else {\n" + elseStmts
                    .mapNotNull { it as? Node.Stmt.Expr }
                    .joinToString("\n") {
                        translateBody(it.expr, parent, elseStmts.size, depth + 1)
                    } + "\n" + tab(depth) + "}"
            }
            return if (childrenCount == 1) wrapInVStack(s, false, depth) else s
        }
        is Node.Expr.BinaryOp -> {
            if (((expr.rhs as? Node.Expr.Call)?.expr as? Node.Expr.Name)?.name == "forEach") {
                isForEach = true
            } else {
                var s = ""
                val lambda = (expr.rhs as? Node.Expr.Call)?.lambda ?: return s
                val variable = lambda.func.params
                    .firstOrNull()?.vars
                    ?.firstOrNull()?.name ?: return s
                s += tab(depth) + "if let $variable = ${expr.lhs.tag} {\n"
                val stmts = lambda.func.block?.stmts ?: return s
                s += stmts
                    .mapNotNull { it as? Node.Stmt.Expr }
                    .joinToString("\n") {
                        translateBody(it.expr, parent, stmts.size, depth + 1)
                    }
                s += "\n" + tab(depth) + "}"
                return s
            }
        }
    }
    var s = ""
    val call = (if (isForEach) (expr as? Node.Expr.BinaryOp)?.rhs else expr)
        as? Node.Expr.Call ?: return s
    val name = (call.expr as? Node.Expr.Name)?.name ?: return s
    val translatedName = translateName(name)
    val arguments = getArguments(call, name).toMutableList()
```

```
206     val modifiers = getModifiers(call).toMutableList()
207     val children = getChildren(call, name)
208     val shouldIgnore = ignoredViews.contains(name)
209 
210     arguments.reversed().forEach {
211         if (modifierArguments.contains(it.name)) {
212             modifiers.add(0, Pair(it.name ?: "", it.expr))
213             arguments.remove(it)
214         }
215     }
216 
217     if (name == "Tab") { return tab(depth) + (call.args
218         .firstOrNull()?.expr as? Node.Expr.Brace)?.block?.stmts
219         ?.firstOrNull()?.tag as? String ?: "" }
220 
221     if (!shouldIgnore) {
222         s += tab(depth) + translatedName
223         if (isForEach) {
224             val argument = call.lambda?.func?.params
225                 ?.firstOrNull()?.vars
226                 ?.firstOrNull()?.name ?: "it"
227             val list = ((expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Name)?.name ?: return s
228             s += "($list, id: \\.self) { $argument in\n"
229         }
230         s += if (children.isEmpty() || arguments.isNotEmpty()) arguments
231             .joinToString(", ","(", ")") {
232                 translateArgument(name, it)
233             } else ""
234     }
235     if (children.isNotEmpty()) {
236         var childrenS = ""
237         s += if (!shouldIgnore && !isForEach) " {\n" else ""
238         if (name == "TabRow") {
239             val items = call.args.firstOrNull { it.name == "items" } ?: return s
240             childrenS += tab(depth + 1) + "ForEach(${items.expr.tag}, id: \\.self) {" +
241                 " ${call.lambda?.func?.params
242                     ?.getOrNull(1)?.vars
243                     ?.firstOrNull()?.name} in\n"
244         }
245         childrenS += children
246                 .joinToString("\n") {
247                     translateBody(
248                         it.expr,
249                         name,
250                         children.size,
251                         depth + if (name == "TabRow") 2 else if (!shouldIgnore) 1 else 0
252                     )
253                 }
254         childrenS += if (name == "Scaffold") "\n" + tab(depth + 2) +
255             ".navigationBarTitle(Text(${getTitle(call)}))" else ""
256         if (name == "TabRow") childrenS += "\n" + tab(depth + 1) + "}"
257         childrenS += if (!shouldIgnore) "\n" + tab(depth) + "}" else ""
258         if (isForEach && children.size > 1) childrenS = wrapInVStack(childrenS, false, depth)
259         s += childrenS
260     }
261     val shouldInsertSpacer = modifiers.removeAll { it.first == "weight" }
262     s += if (name == "Clickable") "\n" + tab(depth + 1) +
263         ".buttonStyle(PlainButtonStyle())" else ""
264     s += if (name == "IconToggleButton") "\n" + tab(depth + 1) +
265         ".padding()" else ""
266     s += if (name == "Icon") "\n" + tab(depth + 1) +
267         ".imageScale(.large)" else ""
268     s += if (name == "Image") "\n" + tab(depth + 1) +
269         ".resizable()" else ""
270     s += if (name == "Card") "\n" + tab(depth + 1) +
271         ".overlay(RoundedRectangle(cornerRadius: 4).stroke(Color.init(white: 0.25), lineWidth: 1)
    ↪ )" else ""
272     s += if (name == "TabRow") "\n" + tab(depth + 1) +
273         ".pickerStyle(SegmentedPickerStyle())\n" + tab(depth + 1) + ".padding()" else ""
274     s += if (modifiers.isNotEmpty()) modifiers
275         .mapNotNull { translateModifier(it, depth + 1) }
276         .joinToString("") { "\n" + tab(depth + 1) + "." + it } else ""
277     if (shouldInsertSpacer) s += "\n" + tab(depth) + "Spacer()"
278     return s
279 }
280 
```

```
fun translateArgument(name: String, argument: Node.ValueArg): String {
    return when (val expr = argument.expr) {
        is Node.Expr.Name -> {
            when (name) {
                "IconToggleButton" -> if (argument.name == "onBookmark") "action: ${expr.name}"
    ↪ else ""
                "TabRow" -> ""
                "IconButton" -> if (argument.name == "onClick") "action: ${expr.name}" else ""
                else -> {
                    argument.name?.let { return it + ": " + expr.name }
                    expr.name
                }
            }
        }
        is Node.Expr.Brace -> {
            when (name) {
                "Clickable" -> "destination: " + expr.tag as? String
                "AlertDialog" -> "title: " + expr.block?.stmts
                    ?.filterIsInstance<Node.Stmt.Expr>()
                    ?.joinToString {
                        translateBody(it.expr, name, expr.block?.stmts?.size ?: 0, 0)
                    }
                else -> argument.tag as? String ?: ""
            }
        }
        is Node.Expr.StringTmpl -> translateString(expr)
        is Node.Expr.Call -> {
            when (name) {
                "Image", "Icon", "IconToggleButton" -> {
                    val type = (expr.expr as? Node.Expr.Name)?.name ?: name
                    when (type) {
                        "vectorResource" -> "systemName: \"" + (((expr.args
                            .firstOrNull()?.expr as? Node.Expr.BinaryOp)?.rhs as? Node.Expr.Name)
    ↪ ?.name ?: expr.args
                            .joinToString { "${it.tag}" }) + "\""
                        "imageResource" -> "\"" +
                                ((expr.args.firstOrNull()?.expr as? Node.Expr.BinaryOp)?.
                                    rhs as? Node.Expr.Name)?.name + "\""
                        else -> type
                    }
                }
                else -> argument.tag as? String ?: ""
            }
        }
        is Node.Expr.BinaryOp -> {
            when (name) {
                "TabRow" -> "selection: \$${expr.lhs.tag}, label: Text(\"\")"
                else -> expr.tag as? String ?: ""
            }
        }
        else -> expr.tag as? String ?: ""
    }
}

fun translateModifier(modifier: Pair<String, Node.Expr>, depth: Int): String? {
    val (name, expr) = modifier
    if (ignoredModifiers.contains(name)) return null
    return when (name) {
        "padding" -> {
            val call = expr as? Node.Expr.Call ?: return null
            val sides = call.args.filter { it.name != null }.map { "." + translateSide(it.name ?:
    ↪  "") }
            "$name(" + (if (sides.isNotEmpty()) (if (sides.size == 1) sides
                .joinToString { it } else "$sides") + ", " else "") +
                (((call.args.firstOrNull()?.expr as? Node.Expr.BinaryOp)
                    ?.lhs as? Node.Expr.Const)?.value ?: "10") + ")"
        }
        "style" -> {
            val category = ((expr as? Node.Expr.BinaryOp)?.rhs as? Node.Expr.Name)
                ?.name ?: return expr.tag as? String ?: ""
            translateTextModifier(category, depth)
        }
        "color" -> "foregroundColor(.secondary)"
        "preferredSize" -> {
            val sizes = (expr as? Node.Expr.Call)?.args
                ?.mapNotNull { ((it.expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Const)?.value
```

```
    ↪ }
                ?: return expr.tag as? String
            "frame(width: ${sizes.getOrElse(0) { "0" }}, height: ${sizes.getOrElse(1) { "0" }})"
        }
        "preferredHeight" -> "frame(height: ${(((expr as? Node.Expr.Call)?.args
            ?.firstOrNull()?.expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Const)?.value ?: "
    ↪ 100"})"
        "preferredWidth" -> "frame(width: ${(((expr as? Node.Expr.Call)?.args
            ?.firstOrNull()?.expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Const)?.value ?: "
    ↪ 100"})"
        "preferredHeightIn" -> "frame(minHeight: ${(((expr as? Node.Expr.Call)?.args
            ?.firstOrNull()?.expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Const)?.value ?: "
    ↪ 100"})"
        "preferredWidthIn" -> "frame(minWidth: ${(((expr as? Node.Expr.Call)?.args
            ?.firstOrNull()?.expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Const)?.value ?: "
    ↪ 100"})"
        "fillMaxSize" -> "frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: .
    ↪ infinity)"
        "fillMaxWidth" -> "frame(minWidth: 0, maxWidth: .infinity)"
        "fillMaxHeight" -> "frame(minHeight: 0, maxHeight: .infinity)"
        "scaleFit" -> "scaledToFit()"
        "clip" -> "clipped()\n" + tab(depth) + ".cornerRadius(${((((expr as? Node.Expr.Call)?.
    ↪ args
            ?.firstOrNull()?.expr as? Node.Expr.Call)?.args
            ?.firstOrNull()?.expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Const)?.value ?:
    ↪ 10})"
        "maxLines" -> "lineLimit(${(expr as? Node.Expr.Const)?.value ?: "nil"})"
        "shape" -> "clipShape(${(expr as? Node.Expr.Name)?.name ?: expr.tag}())"
        "onValueChange" -> "onTapGesture(perform: ${(expr as? Node.Expr.Name)?.name ?: expr.tag})
    ↪ \n"
        "backgroundColor" -> "background(Color.${(expr as? Node.Expr.Name)?.name ?: expr.tag})"
        else -> name
    }
}

fun translateTextModifier(category: String, depth: Int): String {
    return when (category) {
        "subtitle1" -> "bold()\n" + tab(depth) + ".font(.headline)"
        "overline" -> "font(.caption)\n" + tab(depth) + ".fontWeight(.light)"
        "body2" -> "font(.subheadline)\n" + tab(depth) + ".fontWeight(.light)"
        "h6" -> "bold()\n" + tab(depth) + ".font(.system(size: 20))"
        "h4" -> "bold()\n" + tab(depth) + ".font(.title)"
        "caption" -> "font(.caption)\n" + tab(depth) + ".fontWeight(.light)"
        else -> category
    }
}

fun translateSide(dir: String): String {
    return when (dir) {
        "start" -> "leading"
        "end" -> "trailing"
        else -> dir
    }
}

fun translateDeclaration(declaration: Node.Stmt.Decl): String {
    var s = ""
    val property = declaration.decl as? Node.Decl.Property ?: return s
    ((property.expr as? Node.Expr.Call)?.expr as? Node.Expr.Name)?.name?.let {
        when (it) {
            "state" -> s += "@State private "
        }
    }
    s += (if (property.readOnly) "let" else "var") + " " + property.vars
        .filterIsInstance<Node.Decl.Property.Var>()
        .joinToString { it.name } + " = "
    val expression = property.expr ?: return s
    s += when (expression) {
        is Node.Expr.Const -> expression.value
        else -> {
            (property.expr as? Node.Expr.Call)?.lambda?.let {
                it.func.block?.tag
            } ?: run {
                property.expr?.tag
            }
        }
```

```
421     }
422     return s
423 }
424 
425 fun translateProperty(property: Node.Decl.Func.Param): String {
426     var s = "let "
427     val piece = (property.type?.ref as? Node.TypeRef.Simple)?.pieces
428         ?.firstOrNull() ?: return s + property.tag
429     s += property.name + ": " + if (piece.name == "List") "[${(piece.typeParams
430         .firstOrNull()?.ref as? Node.TypeRef.Simple)?.pieces
431         ?.firstOrNull()?.name ?: ""}]" else piece.name
432     return s
433 }
434 
435 fun translateName(name: String): String {
436     return when (name) {
437         "Scaffold" -> "NavigationView"
438         "Crossfade" -> "TabView"
439         "VerticalScroller" -> "ScrollView"
440         "HorizontalScroller" -> "ScrollView(.horizontal, showsIndicators: false)"
441         "Column" -> "VStack(alignment: .leading)"
442         "Row" -> "HStack"
443         "Stack" -> "ZStack"
444         "Clickable" -> "NavigationLink"
445         "forEach" -> "ForEach"
446         "IconToggleButton" -> "Button"
447         "Icon" -> "Image"
448         "Card" -> "Group"
449         "TabRow" -> "Picker"
450         "Box" -> "Group"
451         "Tab" -> "Text"
452         "Toggleable" -> "Group"
453         "IconButton" -> "Button"
454         "AlertDialog" -> "Alert"
455         else -> name
456     }
457 }
458 
459 fun translateTabItem(tabItem: Node.Expr.Call, depth: Int): String {
460     var s = ""
461     s += translateBody(tabItem, null, 1, depth - 1) + "\n"
462     s += tab(depth) + ".tabItem {\n"
463     s += tab(depth + 1) + "Image(systemName: \"house\")\n"
464     s += tab(depth + 1) + "Text(\"${(tabItem.expr as? Node.Expr.Name)?.name}\")\n"
465     s += tab(depth) + "}"
466     return s
467 }
468 
469 fun translateString(string: Node.Expr.StringTmpl): String {
470     var s = ""
471     s += string.elems.joinToString("") {
472         when (it) {
473             is Node.Expr.StringTmpl.Elem.Regular -> it.str
474             is Node.Expr.StringTmpl.Elem.LongTmpl -> "\\(" + it.expr.tag + ")"
475             else -> ""
476         }
477     }
478     return "\"" + s + "\""
479 }
480 
481 fun getArguments(call: Node.Expr.Call, name: String): List<Node.ValueArg> {
482     if (ignoredParametersForViews.contains(name)) return emptyList()
483     return call.args.filter { !((it.tag as? String)
484         ?.startsWith("modifier", true) ?: false) && !ignoredArguments
485         .contains(it.name) && !ignoredArguments
486         .contains((it.expr as? Node.Expr.Name)?.name) && !ignoredParameters
487         .contains(((it.expr as? Node.Expr.BinaryOp)?.lhs as? Node.Expr.Name)?.name)}
488 }
489 
490 fun getChildren(call: Node.Expr.Call, name: String): List<Node.Stmt.Expr> {
491     when (name) {
492         "Scaffold" -> {
493             val bodyContent = ((call.args
494                 .firstOrNull { it.name == "bodyContent" })?.expr as? Node.Expr.Brace)?.block?.
    ↪ stmts
495                 ?.filterIsInstance<Node.Stmt.Expr>() ?: emptyList()
```

```
            val bottomAppBar = ((call.args
                .firstOrNull { it.name == "bottomAppBar" })?.expr as? Node.Expr.Brace)?.block?.
    ↪ stmts
                ?.filterIsInstance<Node.Stmt.Expr>() ?: emptyList()
            return bodyContent + bottomAppBar
        }
        else -> {
            if (viewsWithNormalChildren.contains(name)) {
                return call.lambda?.func?.block?.stmts
                    ?.mapNotNull { it as? Node.Stmt.Expr } ?: emptyList()
            }
            return emptyList()
        }
    }
}

fun getModifiers(call: Node.Expr.Call): List<Pair<String, Node.Expr>> {

    fun getModifierFromBinaryOp(op: Node.Expr?): List<Node.Expr> {
        (op as? Node.Expr.Call)?.let { return listOf(it) }
        val binaryOp = op as? Node.Expr.BinaryOp ?: return emptyList()
        val token = (binaryOp.oper as? Node.Expr.BinaryOp.Oper.Token)?.token?.name ?: return
    ↪ emptyList()
        return when (token) {
            "ADD" -> listOf(getModifierFromBinaryOp(binaryOp.lhs), getModifierFromBinaryOp(
    ↪ binaryOp.rhs))
                .flatten()
            else -> {
                (binaryOp.lhs as? Node.Expr.BinaryOp)?.let {
                    return listOf(getModifierFromBinaryOp(binaryOp.lhs), getModifierFromBinaryOp(
    ↪ binaryOp.rhs))
                        .flatten()
                }
                listOf(binaryOp.rhs)
            }
        }
    }

    return call.args
            .filter { (it.tag as? String)?.startsWith("modifier", true) ?: false }
            .map { it.expr }
            .filterIsInstance<Node.Expr.BinaryOp>()
            .flatMap { getModifierFromBinaryOp(it)
            .map { Pair(((it as? Node.Expr.Call)?.expr as? Node.Expr.Name)?.name ?: "", it) } }
}

fun getTitle(call: Node.Expr.Call): String {
    val topAppBar = (call.args
        .firstOrNull { it.name == "topAppBar" }?.expr as? Node.Expr.Brace)?.block?.stmts
        ?.firstOrNull() ?: return ""
    val title = (((topAppBar as? Node.Stmt.Expr)?.expr as? Node.Expr.Call)?.args
        ?.firstOrNull { it.name == "title" }?.expr as? Node.Expr.Brace)?.block?.stmts
        ?.firstOrNull() ?: return ""
    val text = (((title as? Node.Stmt.Expr)?.expr as? Node.Expr.Call)?.args
        ?.firstOrNull()?.expr as? Node.Expr.StringTmpl) ?: return ""
    return translateString(text)
}

fun wrapInVStack(s: String, align: Boolean, depth: Int):String {
    return tab(depth + 1) + "VStack" + (if (align) "(alignment: .leading)" else "") + " {\n" + s
        .lines()
        .joinToString("\n" + tab(), tab()) { it } + "\n" + tab(depth) + "}"
}

fun tab(depth: Int = 1): String {
    return (1..depth).joinToString("") { "\t" }
}
```