

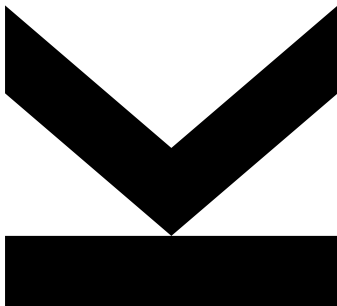
Author
Simon Reitinger

Submission
**Institute of System
Software**

Thesis Supervisor
Prof. Dr. **Herbert Prähofer**

February 2023

Reactive Markup - A Functional UI Library in Haskell



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Computer Science

Sworn Declaration

I hereby declare under oath that the submitted Bachelor's Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, February 2023

Simon Reitingger

Abstract

Creating graphical user interfaces with purely functional programming languages is a challenge. Complex user interfaces tend to manage many different states and have to handle many side effects. However, there exists a suitable design pattern for user interfaces with functional programming languages popular from libraries like *React* or *Jetpack Compose*. In this pattern, functions are used to create a deterministic mapping between the model state and the view state. Such a mapping can be expressed properly in the purely functional language Haskell, where functions per definition have to return the same output for the same input.

Reactive Markup is a library for designing user interfaces where Haskell functions map the model state to the view state. In addition, Reactive Markup incorporates ideas from *Functional Reactive Programming* (FRP), a programming paradigm dealing with state changes happening over time. Together, FRP and view modelling functions are used in order to create a changing graphical user interface in a purely functional programming language. Finally, Reactive Markup has been designed to be polymorphic with regards to the underlying platform, i.e., the same code can be used to create user interfaces for different platforms.

Kurzfassung

Die Erstellung grafischer Benutzeroberflächen mit rein funktionalen Programmiersprachen ist eine Herausforderung. Komplexe Benutzeroberflächen müssen verschiedene Zustände verwalten und haben oft viele Seiteneffekte. Eingeführt in Bibliotheken wie *React* oder *Jetpack Compose*, gibt es hierzu ein Entwurfsmuster, um Benutzeroberflächen mit einem funktionalen Ansatz zu erstellen. Hierbei werden Funktionen verwendet, um eine deterministische Abbildung vom Modellzustand zur grafischen Oberfläche herzustellen. Dies lässt sich gut in der rein funktionalen Programmiersprache Haskell ausdrücken, wo Funktionen die gleiche Ausgabe für die gleiche Eingabe zurückgeben müssen.

Reactive Markup ist eine Bibliothek zur Gestaltung von Benutzeroberflächen mit Haskell-Funktionen, die den Modellzustand auf den Ansichtszustand abbilden. Darüber hinaus enthält Reactive Markup Ideen aus *Functional Reactive Programming* (FRP), einem Programmierparadigma, welches sich mit Zustandsänderungen über der Zeit beschäftigt. FRP und Modell zu Benutzeroberfläche transformierende Funktionen werden gemeinsam verwendet, um eine sich verändernde grafische Benutzeroberfläche in einer rein funktionalen Programmiersprache zu beschreiben. Zusätzlich wurde Reactive Markup so konzipiert, dass es in Bezug auf die verwendete Plattform polymorph ist, d.h., derselbe Reactive Markup Code kann zur Erstellung von Benutzeroberflächen für verschiedene Plattformen verwendet werden.

Contents

| | |
|---|------------|
| Abstract | iii |
| Kurzfassung | iv |
| 1 Introduction and Motivation | 1 |
| 2 Comparison of Java Swing and Jetpack Compose | 3 |
| 2.1 Structure of component code | 3 |
| 2.2 Uni-directional data flow | 4 |
| 2.3 Stateless components | 6 |
| 2.4 Mutation of components | 6 |
| 2.5 Model definition | 7 |
| 2.6 Reacting to changes | 8 |
| 2.7 Change management of Jetpack Compose | 10 |
| 2.8 Celsius/Fahrenheit converter with Jetpack Compose | 12 |
| 3 Reactive Markup | 15 |
| 3.1 Architecture | 15 |
| 3.2 Reactive Markup DSL | 17 |
| 3.2.1 Hello Reactive Markup | 17 |
| 3.2.2 Nested components | 18 |
| 3.2.3 Stateful components | 19 |
| 3.2.4 Dynamic values | 20 |
| 3.2.5 GUI Events | 22 |
| 3.2.6 GUI events changing the model | 25 |
| 3.2.7 Controlling the event flow | 26 |
| 3.2.8 Contexts | 29 |
| 3.2.9 Cross-platform GUIs | 30 |
| 3.2.10 Celsius/Fahrenheit converter | 31 |
| 3.3 Reactive Markup Implementation | 35 |
| 3.3.1 Render typeclass | 35 |
| 3.3.2 The Markup type | 36 |
| 4 Summary and Conclusion | 38 |
| Bibliography | 40 |

Chapter 1

Introduction and Motivation

Graphical user interfaces (GUIs) are inherently stateful, making it difficult to model them with purely functional programming languages. However, there exists a design pattern using functions to design user interfaces which is popular from libraries like *React* or *Jetpack Compose* [5] [2]. The main idea of that design pattern is that the view of an application is defined deterministically by the application model. Given an instance of the application state, it is possible to build the corresponding view. Such a mapping from application state to view can be realised with side-effect free functions.

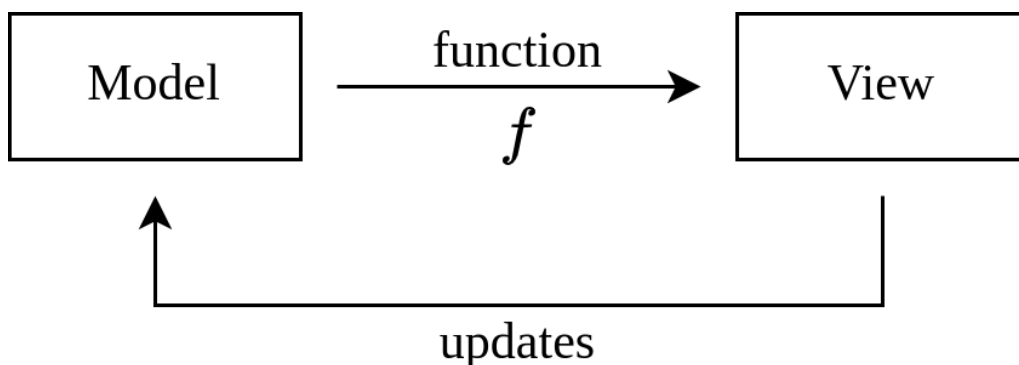


Figure 1.1: Relation of Model and View

Figure 1.1 illustrates the relation between the model and the view. The *Model* represents the current application state and the *View* the corresponding graphical user interface. The function f defines how to build the view based on the model. If events happen, for example button clicks, the view cannot be updated directly. Instead they update only the model. If the model changes, then the function f can be used again to compute the new view.

Listing 1.1 illustrates how a user interface is defined in the popular web framework *React* with normal JavaScript functions. It contains a component which greets the user [5]. The function `Greet` receives the current application state, the `user` as input. The `username` is used to create a personalized message and the view is returned as *HTML*.

Using functions for defining user interfaces is suitable for functional programming languages. Haskell is a purely functional programming language which does not permit

```
function Greet(user) {  
  return (  
    <span>  
      Hello {user.name}!  
    </span>  
  );  
}
```

Listing 1.1: Greeting component with *React*

side-effects [4]. It uses an advanced static type system which supports algebraic data types. The most compelling feature of Haskell is probably that it heavily encourages the use of immutable data structures, making mutability the exception rather than the default [4].

There exist already a few libraries in the Haskell ecosystem which follow the concept in Figure 1.1. One of them is *gi-gtk-declarative* which uses the GTK framework [7] under the hood. Another is *monomer* which is based on the *OpenGL* graphics framework [6]. Both of them use a diffing approach to apply changes: when the model changes and a new view is computed, the old and the new view are compared and the old view is adapted with the minimal set of changes to the new one.

Reactive Markup is a new framework for creating graphical user interfaces in a functional way. In contrast to the previous libraries, Reactive Markup aims to allow for a more fine-grained handling of model changes to avoid diffing. Additionally, Reactive Markup tries to be backend agnostic so that the same code can be used for multiple platforms.

The thesis is structured as follows: Chapter 2 contains a comparison of Java Swing and Jetpack Compose which elaborates on the idea of using functions for GUI applications. This section is included to show the differences between imperative and functional user interfaces. Section 3.1 and Section 3.2 explain some properties of Reactive Markup and illustrate how to use the framework. Finally, Section 3.3 provides a short overview of the internal workings of Reactive Markup. Finally, Chapter 4 concludes the work with a summary and a discussion of open issues.

Chapter 2

Comparison of Java Swing and Jetpack Compose

Java Swing is an example of a library facilitating an imperative approach to GUI programming while *Jetpack Compose* is an example of a declarative framework. In the following, Java Swing and Jetpack Compose are compared to work out the differences between imperative and declarative GUI programming. Due to Jetpack Compose's nature as a declarative library, it provides a higher level of abstraction than Swing. Jetpack Compose components are defined as functions from model to view. In contrast, Swing does have less restrictions on how to organize code and components can be created in various ways.

Java Swing programs tend to be more verbose than Jetpack Compose programs since Jetpack Compose hides more implementation details.

2.1 Structure of component code

Listing 2.1 illustrates a GUI program to horizontally align three labels in Swing. First, a `JPanel` is created as a container for the labels. Next, `JLabels` are created with their text content and immediately appended to the `panel` via the `add` method of `JPanel`. The `JPanel` handles the positioning of the labels and per default uses a flow layout to align them in a row.

Notice that the main way of interacting with GUI components in Swing is via references to those components. It is necessary to first create the `panel` and then later modify its children. With Jetpack Compose, it is possible to create the whole component as one whole structure. Listing 2.2 depicts the same GUI in Jetpack Compose.

```
JPanel panel = new JPanel();  
  
panel.add(new JLabel("Hello"));  
panel.add(new JLabel("from"));  
panel.add(new JLabel("Swing"));
```

Listing 2.1: Components in Swing


```

Row {
    Text("Hello")
    Text("from")
    Text("JetpackCompose")
}

```

Listing 2.2: Component in Jetpack Compose

The outer component is a `Row`, which will align its children in a horizontal manner. The labels are created with the `Text` component. The `Text` components are given to the `Row` as an argument, which adds the `Text` components as children to the `Row`.

In contrast to Java Swing, Jetpack Compose has no references. Parent-child relationships between components are created by handing the children as an argument to the parent. Additionally, it is not possible to later modify components, making components immutable.

In the end, is it a trade-off between flexibility and simplicity. In *Jetpack Compose*, a view component is fully determined by its properties and children at the definition site, not allowing any changes to components in other parts of the application. Swing facilitates explicit mutation of components, supporting more ways to manage components but also increasing code complexity.

2.2 Uni-directional data flow

Jetpack Compose encourages uni-directional data flow. State can be passed down from parents to children, however, children cannot directly influence the state of their parents. Instead, children spawn events which the parent components handle. The parent components decide how to modify their state based on the received events [1].

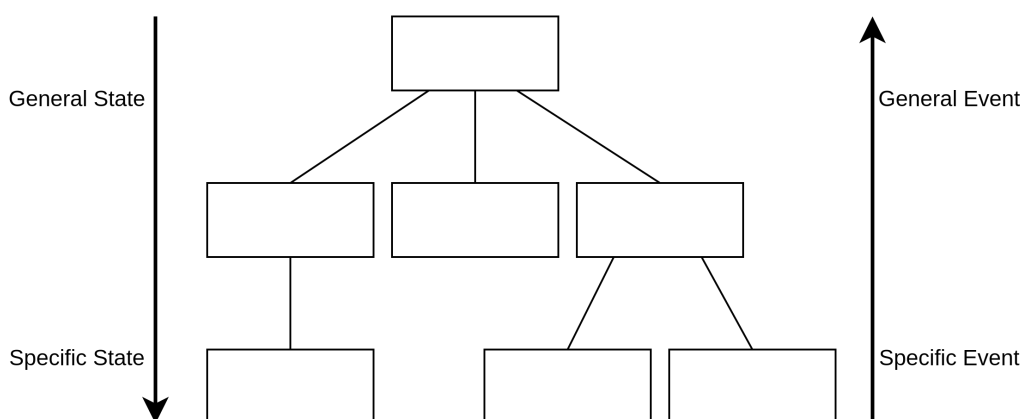


Figure 2.1: Unidirectional Data Flow

Figure 2.1 illustrates the typical data flow in a Jetpack Compose application. Each node represents a component. It is common that components at the leaves deal with more spe-

```

@Composable
fun MyTextField(state: String, onChange: (String) -> Unit){
    OutlinedTextField(
        value = state,
        onChange = onChange
    )
}

```

Listing 2.3: Simple Text field with Jetpack Compose

```

class Model {
    String state
}

public JTextField myTextField(Model model){
    JTextField textField = new JTextField(model.state);
    textField.addActionListener(event -> { model.state = textField.getText()
});
    return textField;
}

```

Listing 2.4: Simple Text field with Java Swing

cific state and events than components closer to the root. For example, a leaf component might be a text field which only cares about its text content. Conversely, a root node might hold the whole application state and coordinates many other components.

Listing 2.3 shows the function for creating a text field in Jetpack Compose. Components in Jetpack Compose are defined as functions with a `@Composable` annotation. Needed state as well as event handlers are given as an input parameter by the parent component when the component is used. In this case, `state` corresponds to the content of the text field and the callback `onChange` is used to handle user inputs. `OutlinedTextField` is a Jetpack Compose component for text fields. Its `value` is set to `state` and `onValueChange` to the function `onChange`. `OutlinedTextField` will take care of rendering the correct text content and executing the `onChange` callback when the user tries to change the text content.

State will often be managed by the parent of a component since children cannot update the state of their parents. In this example, `MyTextField` may not change the text content itself. That content is managed by its parent which will decide how to act when `onChange` is called.

Swing does not enforce any restrictions on the data flow, so state may flow in both directions. Listing 2.4 illustrates how data might flow in both directions in a Swing application. `myTextField` is given the `model` as an input. The `textField` is initialised with the current state of the model. `ActionListeners` for text fields trigger when the *Enter* key is pressed. Therefore, when *Enter* is pressed, the state of the model is directly updated. `myTextField` receives state via the `model` parameter and also modifies the given state without any action from its parent.

```
JTextField textField = new JTextField("Text");  
  
// get content after user updates the text  
String content = textField.getText();
```

Listing 2.5: Reading a Java Swing textfield

```
JLabel label = new JLabel("Text");  
  
// later in program execution  
label.setText("New text");
```

Listing 2.6: Updating a Java Swing label

2.3 Stateless components

Jetpack Compose promotes the idea that the model should define the view in such a way that the view fully depends on the model [2]. The view should not hold any state on its own. For example, the component `MyTextField` shown previously in Listing 2.3 does not contain any state.

The model for the text field is its text content. This text content is explicitly managed by the application and not by the view. In contrast, *Swing* text fields manage their text content on their own, which is illustrated in Listing 2.5. The content of `textField` is managed by itself. When the user changes the text, it can be queried with the method `getText`.

In Java Swing, components have internal state which can be read with functions like `getText` from Listing 2.5. In Jetpack Compose, the state of components can not be queried, i.e., there exists no equivalent function in Jetpack Compose to `getText`. While components in Jetpack Compose may contain internal state, it remains entirely hidden and cannot be accessed. This is a direct consequence of the uni-directional data flow illustrated in Figure 2.1.

2.4 Mutation of components

Modifying the view is straightforward in a Swing application. As long as one holds the reference to a GUI component, one can use its methods to update its state. Listing 2.6 shows that the content of the `label` can be set by using its `setText` method at any point. In contrast, Jetpack Compose components are immutable and cannot be arbitrarily modified. However, the state of Jetpack Compose depends on the state of the application model. Instead of updating the components directly, the application state is modified and the changes are propagated automatically to the components.

Listing 2.7 illustrates how to update the `MyTextField` component from Listing 2.3. `MyTextField` needs the text content it should render and a callback to deal with text

```

@Composable
fun Application(){
    var textContent by remember { mutableStateOf("") }
    MyTextField(textContent, newText => { textContent = newText })
}

```

Listing 2.7: Updating Jetpack Compose Label

```

class Model {
    int intValue = 0;
}

```

Listing 2.8: Model in Swing

input from the user. In order to define state which may change due to user input, Jetpack Compose exports `mutableStateOf`. In this example, `mutableStateOf("")` is used to create a mutable string which is then passed to `MyTextField` as its text content. When the user inputs new text, `textContent` is overwritten with that new text. The `remember` function is needed to automatically propagate updates from `textContent` to `MyTextField`.

The `Application` component follows the pattern illustrated in Figure 1.1 where `textContent` is the model, `MyTextField` is the view function and `newText => { textContent = newText }` is used to update the model.

2.5 Model definition

Assume that the model is a single integer. Defining a model for use with a *Swing* application is straightforward and in line with normal Java practices. The integer is simply represented as an `int` field, as shown in Listing 2.8.

Defining a model for *Jetpack Compose* is more complicated. Each field of the model must be wrapped within a `MutableState`. `MutableState` is a class from Jetpack Compose which is needed to automatically update the GUI on model changes. In Listing 2.9, the `intValue` is wrapped with a `MutableState` which tracks the changes happening to the wrapped value. In order for the `MutableState` to recognize changes, it needs to be directly updated.

Listing 2.10 illustrates an incorrect use of `MutableState`. If the list within `MutableState` is modified, then no changes are detected. In order to update `todos`, one would need to create a new list and set `todos` directly. Instead of wrapping `List` within `MutableState`, it is better to use a `MutableList` provided by Jetpack Compose. The `MutableList` shown in Listing 2.11 can be treated like any other list and supports updates to its elements.

```

class Model {
    val intValue: MutableState<Int> = mutableStateOf(0)
}

```

Listing 2.9: Model in Jetpack Compose

```
class Model {
    val todos: MutableState<List<String>> =
        mutableStateOf(listOf("Gardening", "Cleaning"))
}
```

Listing 2.10: Incorrect use of MutableState

```
class Model {
    val todos: MutableList<String> = mutableListOf("Gardening","Cleaning")
}
```

Listing 2.11: Use MutableList for a list state

Contrary to Jetpack Compose, Swing does not impose any restrictions on model definition, therefore all kinds of classes may be used within the model. In Swing, the model for todos can be done with a normal list as shown in Listing 2.12.

Dealing with more complex model states is not as simple in Jetpack Compose as in Swing since the special container types provided by Jetpack Compose must be used. This may be especially problematic if Jetpack Compose does not support the container that would fit the current situation.

2.6 Reacting to changes

While Swing is a less rigid framework as Jetpack Compose and offers flexibility to the developer, it also means that more code is needed to do the same task. Swing does not have any mechanism for automatically updating the GUI based on Model changes. Therefore, adjusting the GUI to the model needs to be done by hand. One of the most common techniques for synchronizing the model and the GUI is the *event-listener* pattern.

Listing 2.13 provides the code for a model which includes the capabilities for recognizing changes and reacting to them. GUI code can register listeners which update the corresponding GUI element with the new value. In addition to `intValue`, there exists a field for `listeners` which is a list of `IntValueChangeListeners`. An `IntValueChangeListener` is an interface that is called when the `intValue` is changed with `setIntValue`. `IntValueChangeListener` can be added via `addIntValueChangeListener`.

With this mechanism in place, it is possible to create a GUI which will update when the model changes. Listing 2.14 creates a GUI which shows the `intValue` from Listing 2.13 and increases it by one for each button click. The `label` is created with the initial value of the model. Then, a callback is added via `model.addIntValueChangeListener` to update

```
class Model {
    List<String> todos = List.of("Gardening", "Cleaning");
}
```

Listing 2.12: Normal lists as a model for Java Swing

```
class Model {
    int intValue = 0;

    private List<IntValueChangeListener> listeners = new ArrayList<
    IntValueChangeListener>();

    public int getIntValue(){
        return intValue;
    }

    public setIntValue(int intValue){
        this.intValue = intValue;
        for (IntValueChangeListener listener : listeners){
            listener.valueChanged(intValue);
        }
    }

    public addIntValueChangeListener(IntValueChangeListener listener){
        listeners.add(listener)
    }

    public static interface IntValueChangeListener extends EventListener {
        public void valueChanged(int newValue);
    }
}
```

Listing 2.13: Change management in Swing

```
public JPanel ModelValue(Model model){
    JPanel panel = new JPanel();

    JLabel label = new JLabel("IntValue: " + model.getIntValue());
    model.addIntValueChangeListener(newInt ->
        label.setText("IntValue: " + newInt)
    );

    JButton button = new JButton("Click me");
    button.addActionListener(event ->
        model.setIntValue(model.getIntValue() + 1)
    );

    panel.add(label);
    panel.add(button);

    return panel;
}
```

Listing 2.14: Reacting to changes in Swing

```

@Composable
fun ModelValue(model: Model){
    let intValue by remember { model.intValue }
    Column {
        Text("IntValue: " + intValue)
        Button(onClick = { intValue.value++ })
    }
}

```

Listing 2.15: Reacting to changes in Jetpack Compose

```

@Composable
fun Counter(){
    var counter by remember { mutableStateOf(0) }
    Text("Counter at: " + counter)
    Button(onClick = { counter++ }) {}
}

```

Listing 2.16: Simple Counter with Jetpack Compose

the label content when the `intValue` changes. The `button` executes `model.setIntValue` with an increased value whenever it is clicked. Finally, both the `label` and the `button` are added to a `JPanel`.

The same task can be achieved more easily in Jetpack Compose. The example from Listing 2.9 is sufficient as the model. Since `MutableState` tracks changes already, it is not necessary to write additional logic to synchronize model and GUI.

Listing 2.15 defines a Jetpack Compose GUI which is equivalent in functionality to the Java Swing GUI in 2.14. First, `remember` is used to access the current state of the model. Then, the `Text` and `Button` can use the `intValue`. Updating the GUI on model changes with a callback like with `model.addIntValueChangeListener` is not required since Jetpack Compose does this automatically.

2.7 Change management of Jetpack Compose

To synchronize the GUI in Swing, it was necessary to:

1. Provide an update function to apply the necessary changes as in Listing 2.14
2. Provide methods for detecting changes and acting on them as in Listing 2.13

To understand how Jetpack Compose components are updated, consider the following component `Counter`. It comprises a mutable state starting at `0`, `Text` component to render the value of the mutable state and a `button` which increases the state by 1 on every click.

When executing component functions like `Text`, they are automatically added to the current UI container without needing to add them manually as is the case for Swing (cf. Section 2.1). To facilitate this functionality, Jetpack Compose uses a compiler plugin which transforms all functions with a `@Composable` annotation, introducing additional

```

fun Counter(composer: Composer){
    var counter by composer.remember { mutableStateOf(0) }
    composer.add(Text("Couter at: " + counter))
    composer.add(Button(onClick = { counter.value++ })) { Text("Click me") }
}

```

Listing 2.17: Introducing compiler transformations (pseudo code)

```

fun Counter(composer: Composer){
    var counter by composer.remember { mutableStateOf(0) }
    composer.add(
        // Create Component
        Text(),
        // Update Component
        { label => if(composer.changed(counter)){
            label.text = counter
        }});

    // Button never changes
    composer.add(Button(onClick = { counter.value++ }))
}

```

Listing 2.18: Efficient Updates (pseudo code)

parameters and injecting the necessary machinery. Listing 2.17 illustrates the changes made by the compiler plugin (simplified, not conforming to the real generated code).

The `Composer` holds a cache which stores all UI components as well as their relationships. Executing `Counter` for the first time, `composer.remember { mutableStateOf(0) }` stores the initial value of the `MutableState`, which is `0`, within the cache. Next, the `Text` and `Button` components are added and rendered to the screen. When reevaluating `Counter` at the same cache position, the `counter` variable accesses the value within the cache, allowing `Counter` to have local state. The previous versions of `Text` and `Button` are also updated with their new versions created with the new value of `counter`. This updating process of components needs to be efficient since the `Counter` function might get executed many times until the program closes. That is why the compiler plugin performs some analysis on the usage of mutable states like `counter` and introduces a diffing mechanism.

Listing 2.18 is pseudo-code but depicts how efficient updating may be achieved in principle. Contrary to listing 2.17, `composer.add` is not only given the component itself but also a closure on how to update the existing component. The first evaluation of `Counter` in 2.18 is similar as in 2.17, storing the state of `counter` and the two components `Text` and `Button` in the cache. However, subsequent executions will not need to recreate the component every time but will rather only run the closure used for updating. Thus, the text of the cached `Text` component is directly overwritten if the `counter` value has been changed. An update for `Button` is not necessary since it does not depend on any mutable state and therefore the cached `Button` can be used without any additional computation.

It should now be clear how the *Jetpack Compose* framework automatically updates components efficiently. However, it is also important to determine when the update should occur. Listing 2.19 illustrates in pseudo code how Jetpack Compose can recognize changes.


```
fun Counter(composer: Composer){
    var counter by composer.remember { mutableStateOf(0) }
    composer.add(
        // Create Component
        Text(),
        // Update Component
        { label => if(composer.changed(counter)){
            label.text = counter
        }});

    // Button never changes
    composer.add(Button(onClick = { counter.value++; composer.recompose();
    })))
}
```

Listing 2.19: Efficient Updates (pseudo code)

The compiler plugin modifies the callback which is executed on a `Button` click. Whenever the mutable state counter is updated, `composer.recompose()` triggers a reevaluation of Counter at the correct cache position.

2.8 Celsius/Fahrenheit converter with Jetpack Compose

In this section, a Jetpack Compose application which converts from Celsius to Fahrenheit and vice versa is shown. The user can either enter the temperature in Celsius or in Fahrenheit into text fields and the other unit is calculated by the converter.

Listing 2.20 contains the code for the model. The class `Model` stores the current temperature in Celsius. It also contains the functions `celsiusToFahrenheit` and `fahrenheitToCelsius` to convert between Celsius and Fahrenheit. The properties `celsius` and `fahrenheit` are used to get the current temperature from the model. The functions `setCelsius` and `setFahrenheit` create a new `Model` with the given temperature.

Listing 2.8 shows the Jetpack Compose code for the application. The `main` function calls `application`, which runs the `App` component wrapped within a `Window` component. The `App` component holds the current model state which is shown by the two `NumberInputs` as Celsius and Fahrenheit respectively. The `NumberInput` is a text field which only triggers the `onChange` callback if the user enters a valid integer number. Therefore, the callbacks for both `NumberInputs` update the `model` within `App` only when the user enters a valid number.

Figure 2.2 depicts the temperature converter.

```

class Model(val celsius: Int){
    private fun celsiusToFahrenheit(celsius: Int): Int {
        return (celsius * 9 / 5) + 32
    }

    private fun fahrenheitToCelsius(fahrenheit: Int): Int {
        return (fahrenheit - 32) * 5 / 9
    }

    fun setCelsius(celsius: Int): Model {
        return Model(celsius)
    }

    fun setFahrenheit(fahrenheit: Int): Model {
        return Model(fahrenheitToCelsius(fahrenheit))
    }

    val fahrenheit: Int
        get() = celsiusToFahrenheit(celsius)
}

```

Listing 2.20: Celsius/Fahrenheit converter

```

fun main() = application {
    Window(onCloseRequest = ::exitApplication) {
        App()
    }
}

@Composable
@Preview
fun App() {
    var model by remember { mutableStateOf(Model(0)) }
    Column {
        NumberInput("Celsius", model.celsius, {celsius -> model = model.
setCelsius(celsius)})
        NumberInput("Fahrenheit", model.fahrenheit, {fahrenheit -> model
= model.setFahrenheit(fahrenheit)})
    }
}

@Composable
fun NumberInput(label: String, number: Int, onChange: (Int) -> Unit){
    OutlinedTextField(
        value = number.toString(),
        onValueChange = { value -> try {
            onChange(value.toInt())
        } catch(e: NumberFormatException) {
            // do nothing for non-numbers
        }
    },
    label = { Text(label) }
)
}

```



Figure 2.2: Celsius/Fahrenheit converter with Jetpack Compose

Chapter 3

Reactive Markup

Reactive Markup is a GUI library for the programming language Haskell. Haskell is known as a purely functional language, which means that Haskell code does not have side effects and operates on immutable data structure.

The components of a GUI application change regularly while the application is running. When the user inputs characters into a text field, the text field needs to adapt its state. Because GUI applications often need mutable state, GUI programming can be cumbersome in a language like Haskell which does not support arbitrary mutation of components. In order to combine the Haskell and GUI programming, Reactive Markup has a unique architecture which allows for a side-effect free creation of GUI components.

3.1 Architecture

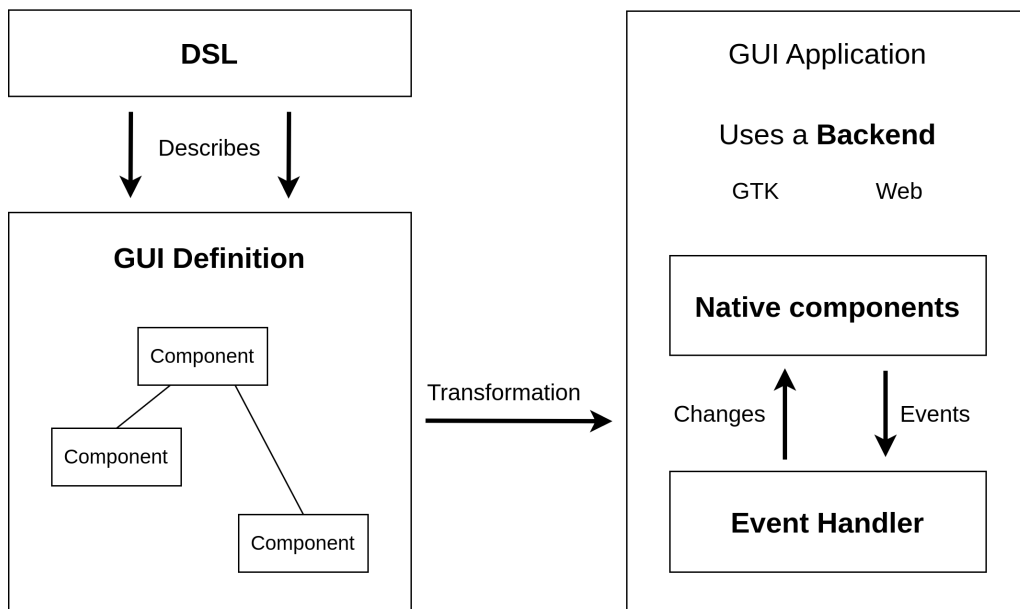


Figure 3.1: Reactive Markup Architecture

The GUI definition is a description of the GUI which should be displayed. This means

```
data Words = Words Text
```

Listing 3.1: The definition for a label component

that the GUI definition does not actually contain the real GUI components, but just the data needed to create them. For example, the *Words* data type in Listing 3.1 contains some text content which is then later used to construct a label which is displayed to the user.

The GUI definition is best thought of as a tree of components descriptions as can be seen in Figure 3.1. As an analogy, the GUI definition is similar to HTML with some additional features to allow for components to change during run-time.

The GUI definition is created by the developer with a *Domain Specific Language (DSL)* and corresponds to the `Markup` datatype. The DSL will be covered in Section 3.2.

Reactive Markup provides a function to transform the GUI definition into an interactive GUI application. Not only are components from the GUI definition mapped to their native counterparts, but also interactive functionality is set up during the transformation. The GUI definition also describes how events are handled and how the GUI updates based on those events. For example, a click on a GTK button might trigger an event handler which was specified in the GUI definition. The event handler may then change a GTK component.

The backend of a Reactive Markup application is the underlying framework used to create the GUI. There exist transformation functions for each backend which transform the GUI definition to a GUI application. The transformation functions map the GUI definition to backend-specific native components and backend-specific event handlers. Currently, there exists support for a GTK and a web backend.

```
gui :: Markup Gtk Paragraph Void
gui = text "Hello Reactive Markup"
```

Listing 3.2: Hello Reactive Markup

3.2 Reactive Markup DSL

In this section the declarative GUI framework Reactive Markup is presented from a user’s perspective. Reactive Markup is a *Domain Specific Language (DSL)* for defining graphical user interfaces. It is a library deeply embedded within the programming language Haskell. Haskell is a lazy functional programming language which is known for its strict management of side effects and its expressive static type system. Reactive Markup makes use of these properties to allow for composition of components and to guarantee that incorrect uses of components are caught at compile time.

Additionally, Reactive Markup draws inspiration from similar declarative GUI libraries and its DSL incorporates the following ideas:

- Components are created as immutable structures.
- Components depend solely on the model such that the GUI itself carries no additional state.
- The GUI is automatically updated whenever the model changes.

These concepts can be realised in the Haskell language due to its favour towards immutable data structures and first-class support for functions. Furthermore, Reactive Markup extensively relies on Haskell features to provide the transformation from GUI definition to GUI application in an extensible and safe manner.

3.2.1 Hello Reactive Markup

In Reactive Markup, GUIs are built using normal Haskell expressions representing the GUI components. Listing 3.2 illustrates a label with the content "Hello Reactive-Markup". The `text` function is used to create a label. `text` takes some text content as input and produces Markup. Markup corresponds to the GUI definition introduced in Section 3.1. The GUI definition describes the GUI which is later transformed into an actual GUI application.

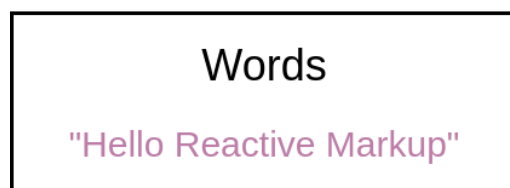


Figure 3.2: Component tree for the Words component

```
nested :: Markup Gtk Common Void
nested = column [
    text "First row",
    text "Second row"
]
```

Listing 3.3: Nested components

Figure 3.2 shows a visualization of the GUI definition of `gui` as a tree of components. The only component within the tree is `Words` from Listing 3.1, which is the component for labels.

The Markup datatype representing the GUI definition has a few type parameters referring to the *backend*, the *context* and the *event* of the component. The *backend* is `Gtk`, which means that `gui` will be used to create a GUI with the *GTK* framework. The *context* is `Paragraph` which defines that `gui` only contains components which are relevant for displaying text. Finally, the *event* is `Void`, indicating that no events can happen since the `Void` type is uninhabited. These parameters will be elaborated upon in the following chapters.

Figure 3.3 shows the created GUI when transforming the markup from Listing 3.2 to a GTK application.

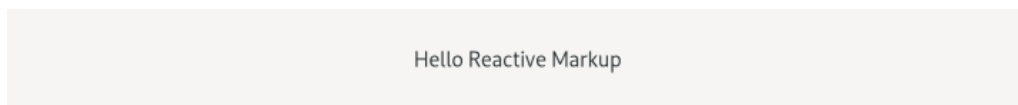


Figure 3.3: Hello Reactive Markup with GTK

3.2.2 Nested components

Markups are regular Haskell expressions which can be freely moved around or saved in variables. Since Markup is just data, it is possible to flexibly transform Markup code in various ways. Markup code can be used as an input to other components which results in natural nesting. This is illustrated in Listing 3.3. The `column` function takes a list of components as input and aligns them vertically. The labels with "First row" and "Second row" are rendered from top to bottom, which can be seen in Figure 3.4.

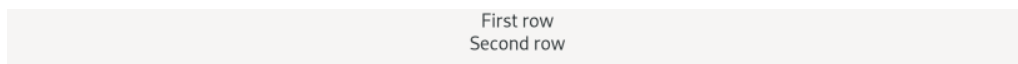


Figure 3.4: Column component in Gtk

Figure 3.5 shows the tree of component for `nested`. The `Column` component is the parent of the two `Words` components since it is higher up in the hierarchy. This means that the `Column` component contains the two `Words` components.

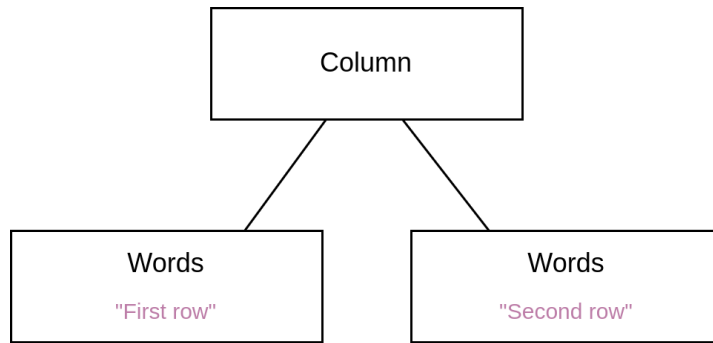


Figure 3.5: Nested component tree

```

data Model = Model Int

statefulGui :: Model -> Markup Gtk Common Void
statefulGui (Model state) = row [
    string "Current state is:",
    string (show state)
]
  
```

Listing 3.4: State as function input

The type parameters of `Markup` have also changed. The *context* of `nested` is now `Common` instead of `Paragraph`. In distinction to `Paragraph` with its focus on text related components, `Common` is a context for all kinds of components with no particular focus.

3.2.3 Stateful components

Reactive Markup follows the design pattern introduced in Figure 1.1 where the view should only depend on the model state. Normal Haskell functions are used to create a mapping between the model state and the view. Using a function for this purpose is also done by other declarative GUI frameworks like *Jetpack Compose*, cf. Section 2.

The GUI definition is a static structure composed of immutable components, which means that it is impossible for components to have internal state. Therefore, the only way to create stateful components is via functions which receive the state as their input parameter and produce a GUI definition as output.

Listing 3.4 depicts an example where the GUI definition depends on the `Model` input parameter. The function `statefulGui` receives the current state of the model as an input parameter and creates a GUI definition using that state. The created view consists of the `row` components with two labels as children. The function `row` works similarly to `column` from Section 3.3 but aligns the children horizontally. The first child of `row` is a constant label while the second one shows the current model state.

Using functions to deterministically create the view from the model state has the advantage that state management is explicit and can be traced throughout the program. Furthermore, the GUI and the model state can never diverge which eliminates the need


```
gui :: Markup Gtk Common Void
gui = statefulGui (Model 7)
```

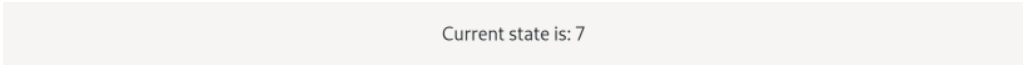
Listing 3.5: Evaluating statefulGui

```
type Behavior a = Time -> a
type Event a = [(Time, a)]
```

Listing 3.6: Behavior and Event

for hand-written synchronization between model and view. Another beneficial property of functions is that using the same model will always produce the same GUI definition.

Evaluating `statefulGui` with the constant model `Model 7` shown in Listing 3.5 results in the GUI in Figure 3.6. If the model were changed to `Model 8`, then it is possible to use `statefulGui` to create a new GUI definition which renders an 8 on the screen.



Current state is: 7

Figure 3.6: statefulGui with 7

3.2.4 Dynamic values

In Listing 3.4 the GUI adjusts depending on the given model. However, the model is fixed when compiling the application and cannot change during runtime. Additional functions are needed to create a GUI definition which can deal with a dynamic model. Reactive Markup takes inspiration from Conal Elliotts work on push-pull based functional reactive programming (*FRP*) [3].

FRP defines semantics to deal with values which change over time with the two concepts `Behavior` and `Event` illustrated with pseudo code in Listing 3.6. A `Behavior` defines a value for each point in time. We can imagine the previously static model to be wrapped within a `Behavior`. Then, as the application is running and advancing in time, the model may change. Contrary to the continuous `Behaviors`, `Events` define values for discrete points in time. A potentially infinite number of events may occur at specific points in time and carry a value as payload [3].

A `Dynamic` in *Reactive Markup* is a combination of both `Behavior` and `Event`. Whenever an `Event` happens, the value of the `Behavior` is updated to the `Event` payload. Figure 3.7 illustrates the life cycle of a `Dynamic`. The `Dynamic` holds an `Int` value. At time t_1 it contains 2. Then, e_1 happens and its value is updated to 1. Later, e_2 occurs and the value is changed to 3. Reading the `Dynamic` at t_2 results in 3.

The `Dynamic` provides a value for each point in time and also provides the discrete points in time when the value changed. By wrapping the model within a `Dynamic`, the model may change during run time and the GUI can be adjusted on changes.

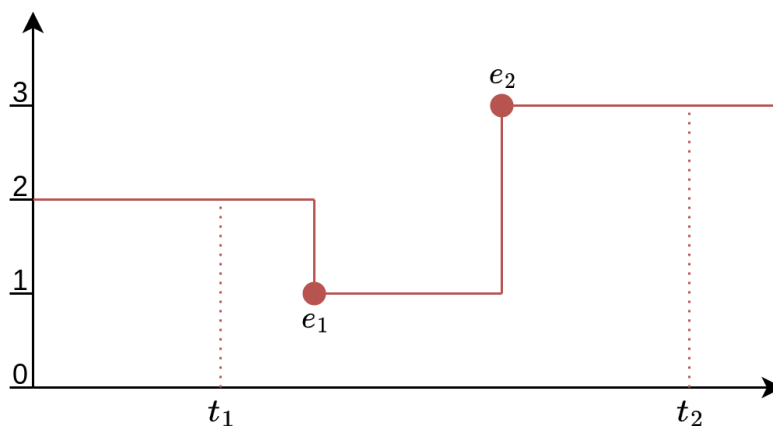


Figure 3.7: Dynamic Int value

```

data Model = Model Int

statefulGui :: Dynamic Gtk Model -> Markup Gtk Common Void
statefulGui state = column [
    "Current state is:",
    dynamicMarkup state modelAsLabel
]
where
    modelAsLabel :: Model -> Markup Gtk Common Void
    modelAsLabel (Model state) = string (show state)

```

Listing 3.7: Model wrapped in Dynamic

Listing 3.7 is similar to Listing 3.4 yet uses a `Dynamic` to wrap the model. Instead of `Model`, `statefulGui` now requires a `Dynamic Gtk Model` as input. The function `dynamicMarkup` can be used to unwrap the `Dynamic` and use the model value. The first argument of `dynamicMarkup` is the `Dynamic` to read. The second parameter is a component using the model value. `dynamicMarkup` responds to changes of the model and updates the GUI automatically. When `state` changes, `modelAsLabel` will be updated as well.

The function `dynamicMarkup` from Listing 3.7 is only a wrapper for the `DynamicMarkup` data type shown in Listing 3.8. `DynamicMarkup` contains dynamic state as well as a function which uses the state. This is sufficient information for the transformation function explained in Figure 3.1 to create an interactive GUI.

Figure 3.8 illustrates the component tree for `statefulGui` where the model is initialised with `Model 7`. `Column` and the left `Words` do not depend on the state; it is only needed for `DynamicMarkup`. At runtime, `DynamicMarkup` reads the current value of the dynamic model and feeds it into `modelAsLabel`. The Component Definition generated by `modelAsLabel` is

```

data DynamicMarkup state backend context event = DynamicMarkup
    (Dynamic backend state)
    (state -> Markup backend context event)

```

Listing 3.8: The `DynamicMarkup` component

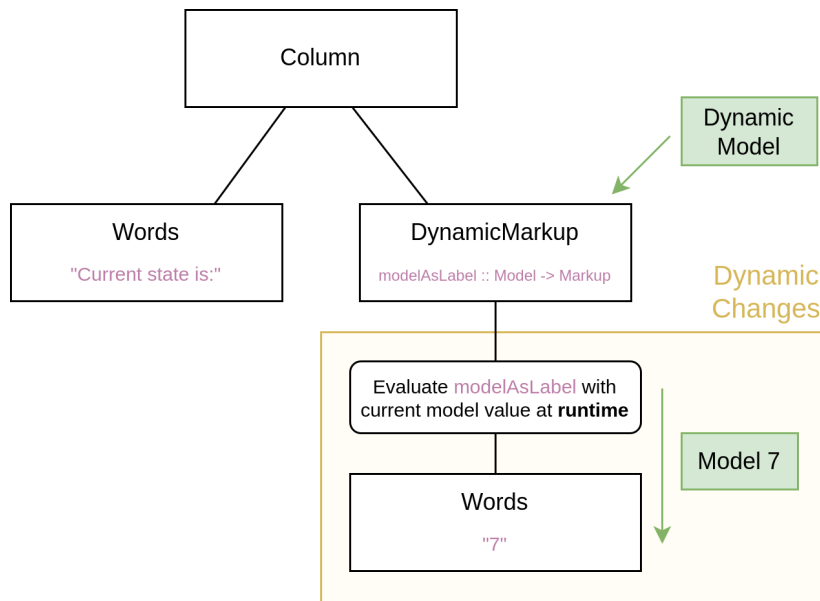


Figure 3.8: Dynamic stateful components

then transformed and dynamically inserted into the GUI application whenever the state changes. The yellow background in the figure is used to signify the parts of the GUI definition which are only generated at run-time.

Although the GUI definition itself is static, the GUI application can be interactive by using Dynamic values. As illustrated in Figure 3.1 about the Reactive Markup architecture, the GUI definition is not only used to create the native components but also to set up the event handler. The components `Words` or `Column` are mapped to native components while `DynamicMarkup` is used to create the event handler which applies changes to the GUI.

Figure 3.9 illustrates what happens when the dynamic model value changes to `Model 8` at run-time. The component `DynamicMarkup` recognizes when changes occur, evaluates `modelAsLabel` with the new model value and updates its child component. Notice that `column` and the `"Current state is:"` component do not need to be reevaluated since they are static and do not depend on the current value of the model. That is why `dynamicMarkup` should be used as late as possible to avoid unnecessary computation. In this example, state changes only result in adjustments of the view for `modelAsLabel`. Since `dynamicMarkup` is used to unwrap a Dynamic only when it is needed, recomputations of the static parts are avoided.

3.2.5 GUI Events

With the help of the `DynamicMarkup` component, it is possible for the GUI to automatically react to changes in the model. However, in order to interactively change the model, it is also necessary to handle user input.

While model changes are directly accessible through Dynamics, GUI events are handled

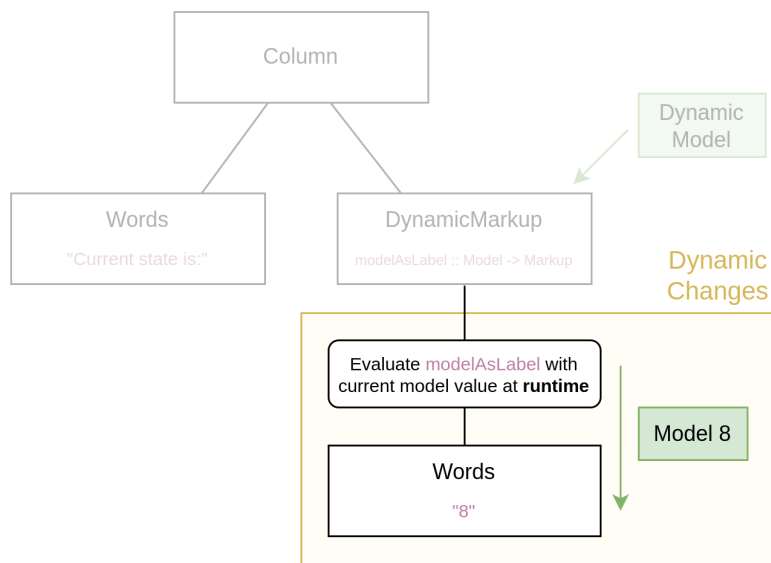


Figure 3.9: Model change at runtime

```
myComponent :: Markup backend context event
```

Listing 3.9: Markup type

more implicitly. The third parameter of the `Markup` type refers to the event as shown in Listing 3.9. The event type indicates that `myComponent` may trigger events of that type. It also means that `myComponent` may not trigger events of any other type as `event`. Therefore, when using `myComponent`, it is exactly known which kinds of events may occur.

Listing 3.10 illustrates a button which emits an event of type `()`. Using `()` as the payload for button clicks events is appropriate since it is enough to know that the click event occurred and no additional information is needed about the event.

The `Button` datatype contains a child `Markup` which is used as the label and which may not fire any events. Additionally, `Button` may contain a `click` value. If the `click` field is `Nothing`, then no event is fired when the button is pressed. However, if `click` is `Just eventPayload`, then an event with `eventPayload` is fired upon button clicks.

In order for the button to fire events, it is necessary to modify the `click` field of the `Button`. However, `button` already wrapped the `Button` within `Markup`. `modifyComponent` can be used to apply a function on the underlying component that is wrapped by `Markup`. `\btn -> btn {click = Just ()}` is a function which sets the `click` field of `Button` it is given to `Just ()`. Therefore, `clickButton` is a button with the label "Click me" where the `click` field was set to `Just ()`. It has the type `Markup Gtk Common ()` and may trigger events of type `()`. The resulting GUI is depicted in Figure 3.11.

Listing 3.11 shows an example of a `column` which contains two `clickButtons`. The function `twoButtons` has an event type of `()` although `column` does not trigger any events. That

```

data Button backend event = Button
  { label :: Markup backend Paragraph Void,
    click :: Maybe event
  }

button :: Markup Gtk Paragraph Void -> Markup Gtk Common Void
button label = Markup
  ( Button
    { label = label,
      click = Nothing
    }
  )

clickButton :: Markup Gtk Common ()
clickButton = modifyComponent (\btn -> btn {click = Just ()}) $
  button (text "Click me")

```

Listing 3.10: Simple Button



Figure 3.10: GTK GUI of clickButton

is because `column` inherits its event type from its children.

Figure 3.11 depicts the GUI definition for `twoButtons`. When one of the `clickButtons` is clicked, its event will automatically be handed to the `column` component. Since the `column` component does not handle events, it passes the event upwards to its parent component. The parent component can either handle the events or once again propagate them upwards.

Since the event type is tracked at the type level, it is ensured at compile time that the automatic propagation of events does not result in any errors when managing events. The type checker is also able to infer the event type of the parent component when the event types of the children are known.

One issue of this approach is that `column` is only able to pass through events of a single type. If there are two different event types, then the solution is to create a sum type which can contain both event types. Listing 3.12 shows how to create an `Event` datatype with two different constructors `EventA` and `EventB`. The buttons fire events of the same type but with different values so that they can later be distinguished in the event handler.

```

twoButtons :: Markup Gtk Common ()
twoButtons = column [clickButton, clickButton]

```

Listing 3.11: Automatic event propagation

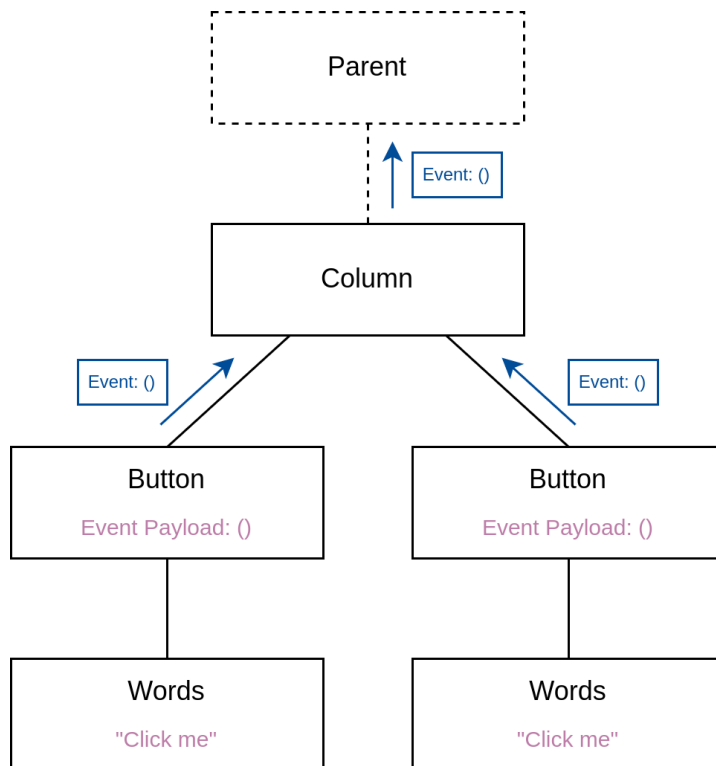


Figure 3.11: Component tree for twoButtons

```

data Event = EventA | EventB

twoButtons :: Markup Gtk Common Event
twoButtons = column [
  modifyComponent (\btn -> btn {click = Just EventA}) $
    button "Fires EventA",
  modifyComponent (\btn -> btn {click = Just EventB}) $
    button "Fires EventB"
]

```

Listing 3.12: Dealing with different event types

3.2.6 GUI events changing the model

The previous section only showed how to trigger events but not how to react to them. Since events occur only at specific points in time, it is not possible to simply get their value. Instead, Reactive Markup provides components which act whenever an event occurs. One of those event-handling components is `simpleLocalState`. It is similar to `remember(mutableStateOf(initialState))` in Jetpack Compose (cf. 2.4). `simpleLocalState` stores a local model and updates it whenever an event occurs.

Listing 3.13 illustrates a counting application with a button and a label. Each click of the button increases the number shown in the label by one. The `countingButton` component is created with the function `simpleLocalState` with the following parameters:

- The first parameter is the `initialState` of the model.

```

data Model = Model Int

clickButton :: Markup Gtk Common ()
clickButton = button (string "Click me") $= \btn -> btn {click = Just ()}

countingButton :: Markup Gtk Common Void
countingButton =
  simpleLocalState
    initialState
    handleButtonClick
    buttonWithNumber
  where
    initialState :: Model
    initialState = Model 0

    handleButtonClick :: () -> Model -> Maybe Model
    handleButtonClick () (Model state) = Just (Model (state + 1))

    buttonWithNumber :: Dynamic Gtk Model -> Markup Gtk Common ()
    buttonWithNumber model = column
      [dynamicMarkup model modelAsLabel, clickButton]

    modelAsLabel :: Model -> Markup Gtk Common ()
    modelAsLabel (Model int) = string $ show int

```

Listing 3.13: Counting application in Reactive Markup

- The second parameter `handleButtonClick` is a function which handles the occurrence of events. Given the event data `()` and the old `Model` state, it produces the new `Model` state by increasing it by one.
- The third parameter `buttonWithNumber` is a component with the `model` as an input parameter. The `model` of type `Dynamic Gtk Model` has the initial value specified by `initialState` and is updated according to `handleButtonClick` whenever a `()` event happens.

The `()` events are triggered by the `clickButton` component from listing 3.10 and are propagated through the `column` component and then handled by `simpleLocalState`.

Figure 3.12 shows the component tree for `countingButton`. When the *Button* is clicked, the `()` event is propagated upwards to *LocalState*. The *LocalState* component evaluates the function `handleButtonClick` with the event payload `()` and the old `Model` to produce the new `Model`. The new `Model` is used to update the *Dynamic Model*. The *Dynamic-Markup* component recognizes when changes happen in *Dynamic Model* and reads the changed value of *Dynamic Model*. The new model value is used to generate a new GUI definition with `modelAsLabel` which is then transformed and inserted into the running GUI application. The resulting GTK application is illustrated in Figure 3.13.

3.2.7 Controlling the event flow

Since events are propagated automatically, there is no need to explicitly use callbacks to manage events within an application. However, not using callbacks also means that events

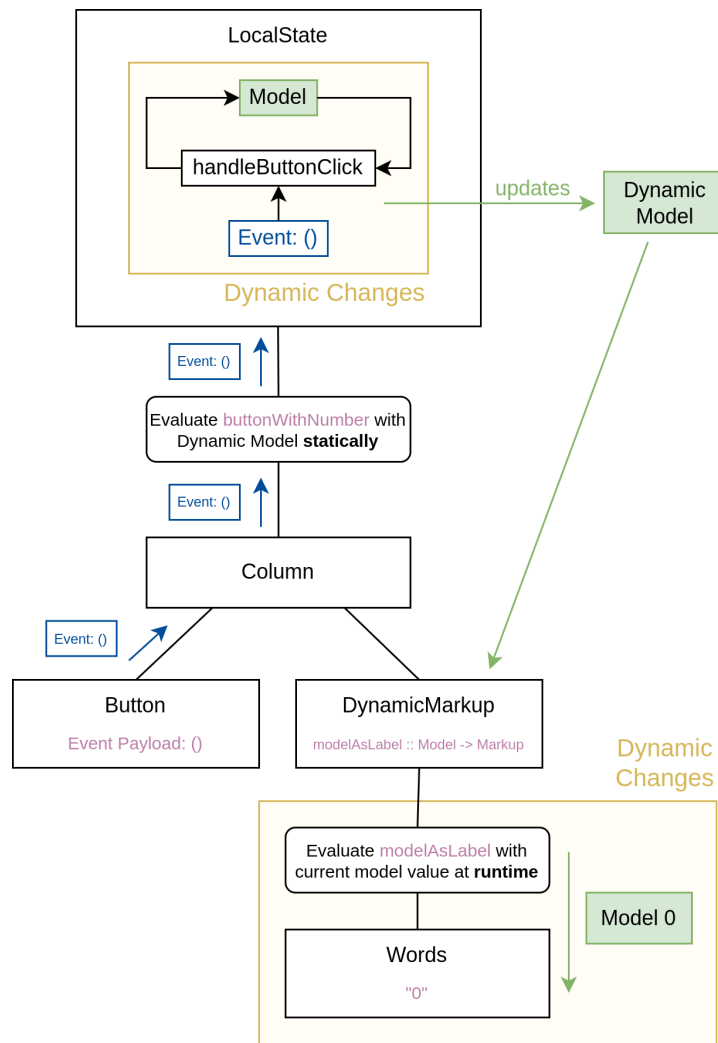


Figure 3.12: GUI definition for countingButton

cannot be accessed as directly, which is why Reactive Markup provides components to modify events.

It is possible to map events as they travel up the component hierarchy. Since Markup implements Functor, `fmap` can be used to modify the event triggered by a component. In Listing 3.14, the already familiar `clickButton` is modified so that `clickButtonText` triggers an event with the value "Button clicked". The original event value `()` of `clickButton` is mapped to "Button clicked" in `(\() -> "Button clicked")`. This mapping is then applied to `clickButton` by using `fmap`.

Events can not only be mapped but also filtered. Listing 3.16 implements a text field component which accepts only lower case letters while Listing 3.15 explains the `TextField`

```
clickButtonText :: Markup Gtk Common Text
clickButtonText = fmap (\() -> "Button clicked") clickButton
```

Listing 3.14: Mapping events from components

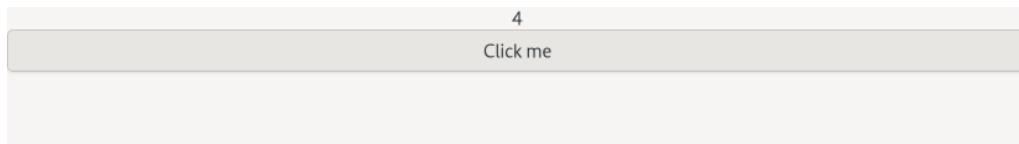


Figure 3.13: Counting button in GTK

```

data TextField t e = TextField
  { text :: Dynamic t Text,
    change :: Maybe (Text -> e)
  }

myTextField :: Dynamic Gtk Text -> Markup Gtk Common Text
myTextField content =
  modifyComponent (\textfield -> textfield {change = Just id}) $
    textField content

```

Listing 3.15: The TextField component

component.

The `TextField` datatype represents a text field. The `text` field stores the dynamic text content which may change over time. The `change` field may contain a function which creates an event from the current text content. If such a function is specified, then the event is created whenever the text of the text field changes.

The function `textField` is a function which wraps the `TextField` datatype within `Markup` and sets the `text` field of `TextField` to `content`. Similar to Listing 3.10, `modifyComponent` is used to access `TextField` and set its `change` field to `Just id`. Since the `change` field is not `Nothing`, an event will be fired. The event is calculated with the `id` function, which means that the text content of the text field will be used directly as the event value. Figure 3.14 shows the text field component in GTK.

Figure 3.14: myTextField in GTK

The function `myTextField` emits events for all content changes, however only text that is lowercase should trigger an event. The component `filterEvents` can filter out events which are not interesting. It is used in Listing 3.16 to only let lowercase words pass through. The first argument of `filterEvents` is a function with type `event -> Maybe event` to decide which events should be kept. Events wrapped within a `Just` are kept while `Nothing` indicates that the event should be dropped. The second argument of `filterEvents` is the component from which the events are filtered, which is `myTextField` from the previous Listing 3.15.

Figure 3.15 illustrates the GUI definition for `lowerCaseTextField`. The `TextField` triggers events whenever its content is changed. The events are propagated to `FilterEvents` which

```

lowerCaseTextField :: Dynamic Gtk Text -> Markup Gtk Common Text
lowerCaseTextField content = filterEvents
  (\content -> allLowerFilter content
    then Just content
    else Nothing
  )
  (myTextField content)
  where
    allLowerFilter text = all isLower (unpack text)

```

Listing 3.16: A text field for lower case inputs

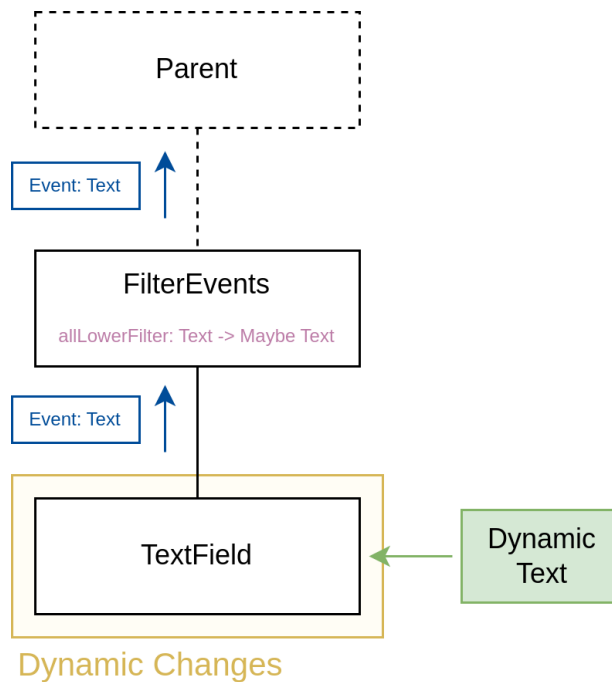


Figure 3.15: GUI definition for lowerCaseTextField

drops all events which do not fulfill the `allLowerFilter` predicate. The events which fulfill the predicate are propagated upwards to the parent.

3.2.8 Contexts

GUI components are often assumed to be used within a specific context. For example, it does not make sense to create a button as a child component of another button. Such mistakes are normally only found at run-time. Depending on the used components, different restrictions or capabilities may apply when defining the GUI.

Contexts are a unique concept in *Reactive Markup* and make it possible to constrain which kinds of components may be contained within a component at compile-time. The context type parameter of Markup shown in Listing 3.17 is used to track the context of a Markup. As mentioned in sections 3.2.1 and 3.2.2, `Paragraph` is used for textual content while `Common` is a general purpose context for all kinds of components with no special

```
myComponent :: Markup backend context event
```

Listing 3.17: Context of Markup

```
myLabel :: Markup Gtk Paragraph Void
myLabel = text "Label"
```

```
myRow :: Markup Gtk Common Void
myRow = row []
```

```
boldLabel = bold myLabel -- Legal
```

```
boldRow = bold myRow -- Illegal
```

Listing 3.18: Bold component is only allowed for paragraphs

restrictions.

Listing 3.18 and Listing 3.19 show where the more specialized `Paragraph` can be used while `Common` can not be used. In Listing 3.18, `myLabel` is a label with a `Paragraph` context while `myRow` is a row with a `Common` context. The `bold` function can only be used on components with the `Paragraph` context since the behavior for boldening `Common` components like a row is not clear. Therefore, only `boldLabel` is legal while trying to compile `boldRow` will result in a compilation error.

Listing 3.19 shows the concatenation of components with a `Paragraph` context via the `Semigroup` typeclass. The `<>` operator can be used to concatenate two `Paragraph` components similar to how strings can be combined. In this example, `myLabel` is a component with a `Paragraph` context. `concatenatedLabels` combines an italic version of `myLabel` and a bold version of `myLabel`. The result can be seen in Figure 3.16.

3.2.9 Cross-platform GUIs


Reusing the same code for different platforms can prove to be rather challenging. Reactive Markup has been created with the idea in mind to support multiple platforms. Ideally, the same code base can be used for various systems without any major modifications.

Reactive Markup facilitates coding for various platforms with the `backend` type parameter of Markup shown in Listing 3.20. The `backend` of a Markup specifies the architecture the GUI is programmed for. Reactive Markup makes sure at compilation-time that only components are used which are supported by the chosen platform.

```
myLabel :: Markup Gtk Paragraph Void
myLabel = text "Label"
```

```
concatenatedLabels :: Markup Gtk Paragraph Void
concatenatedLabels = italic myLabel <> text " " <> bold myLabel
```

Listing 3.19: Concatenated paragraphs



Label Label

Figure 3.16: Concatenated paragraphs in GTK

```
myComponent :: Markup backend context event
```

Listing 3.20: backend of Markup

Up until now, all examples were created with the GTK backend. However, there exists another backend called `RDom` which uses the library `reflex-dom` under the hood to create web interfaces.

Recall the counting button example from before in 3.13. It can be ported to `RDom` with minimal adjustments which are shown in Listing 3.21. The counting button example consists of a label, a button and state management with `simpleLocalState`; all of which are supported by `RDom`. In order to specify that the GUI is for `RDom` instead of `Gtk`, it is sufficient to modify the `backend` parameter of `Markup` so that all occurrences of `Gtk` are replaced with `RDom`.

Listing 3.21 can now be run on the browser. It is identical to Listing 3.13 except that `Gtk` has been replaced with `RDom`. Figure 3.17 depicts the counting button example for the two backends `RDom` and `Gtk`.

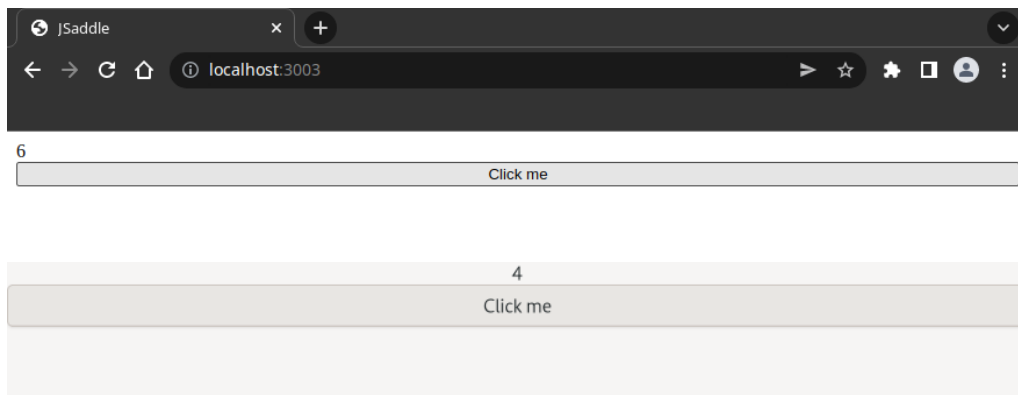


Figure 3.17: Reactive Markup on the browser and in GTK

3.2.10 Celsius/Fahrenheit converter

In this section, a Celsius/Fahrenheit converter is implemented with Reactive Markup. The temperature can be entered in Celsius or in Fahrenheit and the other unit is calculated by the program. It is similar to the converter application built with Jetpack Compose (c.f. 2.8).

Listing 3.22 shows the application model. The `Model` datatype contains the temperature in Celsius and the function `fahrenheit` computes the temperature in Fahrenheit from

```

data Model = Model Int

clickButton :: Markup RDom Common ()
clickButton = button [\buttonOptions -> buttonOptions {click = Just ()}] (
    string "Click me")

countingButton :: Markup RDom Common Void
countingButton =
    simpleLocalState
        initialState
        handleButtonClick
        buttonWithNumber
    where
        initialState :: Model
        initialState = Model 0

        handleButtonClick :: () -> Model -> Maybe Model
        handleButtonClick () (Model state) = Just (Model (state + 1))

        buttonWithNumber :: Dynamic RDom Model -> Markup RDom Common ()
        buttonWithNumber model =
            column
                [ dynamicMarkup model $ \(Model int) -> (string $ show int),
                  clickButton
                ]

```

Listing 3.21: Counting button ported to the web

the `Model`. The functions `setCelsius` and `setFahrenheit` are used to create a new model with the given temperature, exactly like with Jetpack Compose in Listing 2.20.

In contrast to Jetpack Compose, it is also necessary to define which events may happen within the application. `AppEvent` is the datatype used for the events. The `SetCelsius` and `SetFahrenheit` constructors mean that Celsius or Fahrenheit need to be updated. The function `handleEvent` handles an `AppEvent` and computes the `Model`.

Listing 3.23 contains the GUI code needed for the converter. The main function uses `runGtk` to start the Reactive Markup application with the GTK backend. The `simpleApp` function creates a Reactive Markup application given the app name and a `Markup`. `converter` contains the main part of the GUI. The function `simpleLocalState` creates the `Model` as local state which gets updated with the function `handleEvent` when an `AppEvent` happens.

The `numberField` is a text field which only accepts integer values, similar to the Jetpack Compose `NumberInput` from Listing 2.8. It is given the current temperature as a dynamic value since the model changes during program execution. In contrast to the Jetpack Compose solution, `numberField` does not require a callback parameter because events are tracked automatically. To differentiate between events from the two `numberFields`, they are mapped with `SetCelsius` and `SetFahrenheit` respectively. Then it is known in the `handleEvent` function where the user has entered a new number.

Figure 3.18 depicts the Celsius/Fahrenheit converter.

```

data Model = Model { celsius :: Int }

fahrenheit :: Model -> Int
fahrenheit (Model celsius) = ((celsius * 9) `quot` 5) + 32

setCelsius :: Int -> Model
setCelsius celsius = Model celsius

setFahrenheit :: Int -> Model
setFahrenheit fahrenheit = setCelsius $ ((fahrenheit - 32) * 5) `quot` 9

data AppEvent = SetCelsius Int | SetFahrenheit Int

handleEvent :: AppEvent -> Model -> Maybe Model
handleEvent (SetFahrenheit fahrenheit) _ = Just (setFahrenheit fahrenheit)
handleEvent (SetCelsius celsius) _ = Just (setCelsius celsius)

```

Listing 3.22: Model for Celsius/Fahrenheit converter

```

main :: IO ()
main = runGtk $ simpleApp "Temperature Converter" converter

converter :: Markup Gtk Common Void
converter = simpleLocalState (Model 0) handleEvent $ \model ->
  column
  [ "Celsius",
    fmap SetCelsius $ numberField (fmap celsius model),
    "Fahrenheit",
    fmap SetFahrenheit $ numberField (fmap fahrenheit model)
  ]

numberField :: Dynamic Gtk Int -> Markup Gtk Common Int
numberField state =
  let text :: Dynamic Gtk Text = pack . show <$> state
      in filterEvents parseNumber $ modifyComponent (\textField -> textField {
        change = Just id}) $ textField text
  where
    parseNumber :: Text -> Maybe Int
    parseNumber text = readMaybe $ T.unpack text

```

Listing 3.23: Markup for Celsius/Fahrenheit converter

| |
|------------|
| Celsius |
| 100 |
| Fahrenheit |
| 212 |

Figure 3.18: Celsius/Fahrenheit converter with Reactive Markup

```
class Render component backend context where
  render :: forall event. component event ->
    RenderTarget backend context event
```

Listing 3.24: Render typeclass

```
data HtmlBackend

data HtmlText event = HtmlText Text
type instance RenderTarget HtmlBackend Paragraph = HtmlText
```

Listing 3.25: Implementing a HTML backend

3.3 Reactive Markup Implementation

The implementation of Reactive Markup focuses on defining the transform function which maps GUI definitions to actual GUI applications. Recall the Reactive Markup architecture from Figure 3.1. The GUI Definition defined by the DSL is transformed into native GUI code which also handles changes and reacts to events.

3.3.1 Render typeclass

The transform function needs to be defined for every component. In addition, it is also important to consider that each backend needs its own transform functions. In order to represent this relationship, Reactive Markup uses the `Render` typeclass shown in Listing 3.24. The `Render` typeclass implements the transformation function for a given `component`, `backend` and `context`. Given a `component` as input, the `render` function produces a `RenderTarget`. The type of the event is universally quantified so that `render` can be used with all events, no matter which type of events a component may trigger.

The `RenderTarget` is an open type family, which can be used in Haskell to compute type-level terms. In the case of `RenderTarget`, the transformation result type is calculated based on the used `backend`, `context` and the occurring `event`.

With the `Render` typeclass, it is possible to define transformation functions for many components with various backends. Different contexts can also lead to different transformation functions since `context` is a type parameter of `Render`.

Listing 3.25 shows how to implement a backend which generates HTML code for Reactive Markup paragraphs. The type `HtmlBackend` is the name for the backend which generates HTML code. The `RenderTarget` for the backend `HtmlBackend` and for the context `Paragraph` is `HtmlText`. This means that the `render` function will transform Reactive Markup components into `HtmlText`. `HtmlText` is a wrapper over the normal `Text` type. The event parameter of `HtmlText` is unused since this example does not deal with events.

The transform function is implemented as instances of the `Render` typeclass which is shown in Listing 3.26. The `Render` instances define the mapping between the components `Words`, `Bold` and `Combine` and the target type `HtmlText`. `Words` components are mapped


```

instance Render Words HTMLBackend Paragraph where
  render (Words text) = HtmlText text

instance Render Bold HTMLBackend Paragraph where
  render (Bold boldComponent) =
    let (HtmlText htmlText) = render boldComponent
    in HtmlText $ "<b>" <> htmlText <> "</b>"

instance Render Combine Words HTMLBackend Paragraph where
  render (Combine component1 component2) =
    let (HtmlText htmlText1) = render component1
        (HtmlText htmlText2) = render component2
    in HtmlText $ innerText <> htmlText2

```

Listing 3.26: Transform functions for the HTML backend

```

data Markup backend context event = forall widget. Render component backend
  context => Markup (component event)

```

Listing 3.27: The Markup datatype

verbatim to `HtmlText`, `Bold` components are wrapped with `` tags and the `Combine` component is used to concatenate two texts.

Listing 3.26 illustrates how components can be mapped to a specific backend. Writing such transformation functions for other backends like `Gtk` results in different `RenderTargets` but is otherwise similar. Dynamic behavior can also be achieved by choosing an appropriate `RenderTarget`, so this interface is also sufficient for more complex components like `DynamicMarkup` or `LocalState`.

3.3.2 The Markup type

The `Markup` datatype defined in Listing 3.27 may hold any component which implements the `Render` typeclass. Only components which really implement `Render` for the given backend and context can be wrapped by the `Markup` type.

The main use of `Markup` is to aggregate all components which may occur in a specific context and within a specific backend into one type. A sub-typing relationship between components and the `Markup` type is created. In object-oriented terms, `component backend context event` would be a subclass of `Markup backend context event` where `component` can be any component which implements `Render component backend context`.

The `Markup` type can then be transformed by applying the `render` function on the inner component as shown in Listing 3.28. It is known that the wrapped component implements `Render` due to the constraint in the definition of `Markup`. The `RenderTarget` depends on the `backend`, the `context`, the `event`, but not the used component. Therefore, although `Markup` may contain different components, all `Markups` with the same type also have the same `RenderTarget`.

```
renderMarkup :: Markup backend context event -> RenderTarget backend context
              event
renderMarkup (Markup component) = render component
```

Listing 3.28: Rendering Markup

Chapter 4

Summary and Conclusion

Declarative GUI frameworks aim to simplify the creation of user interfaces by increasing the level of abstraction. One such library is *Jetpack Compose*, which uses functions mapping the application model to the view as the primary method to create the GUI. Through a comparison with the imperative *Java Swing* GUI framework, it can be seen that less code is necessary with Jetpack Compose and that Jetpack Compose takes care of some tasks which had to be done manually in Java Swing. For example, updating the view whenever the model changes needs to be done manually in Java Swing while this happens automatically in Jetpack Compose.

Similar to Jetpack Compose, Reactive Markup is also a declarative GUI library which uses functions to define GUI components. It is written in Haskell, therefore Reactive Markup GUIs are defined in a deterministic fashion with immutable data structures and without side-effects. Through the use of dynamic values, it is possible to control the mutable parts of a user interface within an immutable environment.

In contrast to other libraries, Reactive Markup has a strict separation of the immutable GUI definition and the mutable GUI program. Through transformer functions, a GUI definition can be converted to a GUI program. This feature allows the development for multiple platforms as well as the definition of GUIs in a side-effect free manner.

Moreover, Reactive Markup can be used to develop GUI applications for multiple platforms with minimal changes by using different transformer functions. It is also possible to extend Reactive Markup with additional low-level components without modifying the base library. In this manner, platform-specific components can be developed with Reactive Markup.

Reactive Markup is rather flexible. However, this is achieved through the use of many advanced Haskell features which makes it more difficult to use. While the type parameters of the *Markup* type allow for an adaptable GUI depending on the usage situation, they can also be challenging to handle and may be the cause of hard to understand type errors. This is especially a problem when the compiler cannot infer the type by itself and the programmer needs to fill in the correct types by hand. As of right now, Reactive Markup also does not have a sufficient amount of different components to handle a wide variety of GUI application needs. While it is possible to develop basic applications, more specific components like tables or images are not yet available.

In summary, Reactive Markup can already be tested with in small GUI applications. For a wider adoption, more components need to be implemented. Additionally, it would be beneficial if the *Markup* type could be simplified.

Bibliography

- [1] developer.android.com. 2022. Architecting your compose ui. Retrieved 08/16/2022 from <https://developer.android.com/jetpack/compose/architecture>.
- [2] developer.android.com. 2022. Thinking in compose. Retrieved 08/16/2022 from <https://developer.android.com/jetpack/compose/mental-model>.
- [3] Conal Elliott. 2009. Push-pull functional reactive programming. In *Haskell Symposium*. <http://conal.net/papers/push-pull-frp>.
- [4] haskell.org. 2022. Haskell: an advanced, purely functional programming language. Retrieved 12/26/2022 from <https://www.haskell.org>.
- [5] reactjs.org. 2022. Tutorial: intro to react. Retrieved 08/12/2022 from <https://reactjs.org/tutorial/tutorial.html#function-components>.
- [6] Francisco Vallarino. 2022. Monomer. Retrieved 12/28/2022 from <https://github.com/fjvallarino/monomer>.
- [7] Oskar Wickström. 2022. Gi-gtk-declarative. Retrieved 12/28/2022 from <https://owickstrom.github.io/gi-gtk-declarative>.