University of Dublin

TRINITY COLLEGE

GUI Support for U·(TP)²

Jeremy Barisch Rooney

B.A. (Mod.) Integrated Computer Science

Final Year Project May 2017

Supervisor: Dr. Andrew Butterfield

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

Declaration

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

Jeremy Barisch Rooney

Date

Permission to lend

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

Jeremy Barisch Rooney

Date

Abstract

U·(TP)² is an existing computer application which is a Theorem Proving Assistant for the Unifying Theory of Programming. The application is written in Haskell, with a graphical user interface (GUI) built with the mature WxHaskell library.

We attempt to develop a second GUI for U·(TP)² using a Haskell library named Threepenny-gui. Threepenny-gui provides a more consistent experience across operating systems by using the web browser as a display, and promotes a more functional style of writing a GUI via functional reactive programming.

Threepenny-gui is a young library in the Haskell GUI space. In using it to build a GUI for U·(TP)² we realise both its potential but also discover some limitations, contribute to its source, and publish Haskell packages which provide extensions to Threepenny-gui.

Acknowledgements

Acknowledge the various people here

Contents

Part I

Background

Chapter 1

Existing Software

U·(TP)² is an existing computer application which is a Theorem Proving Assistant for the Unifying Theory of Programming. ”Theorem Proving Assistant” means it can be used to assist the development of theorems, the theorems in question are related to the ”Unifying Theory of Programming”. The application is written in Haskell, with a graphical user interface (GUI) built with the mature wxHaskell library. U·(TP)² has been in development since at least March 2010 which is when the source originally appears on BitBucket. U·(TP)² was formerly known as Saoithín.

WxHaskell is a GUI library for Haskell that was started in July 2003 REF but moved to its current repository in January 2007 REF when the project was taken over by a new set of maintainers. The goal of the project is to provide an ”industrial strength GUI library” for Haskell REF. The wxHaskell team attempt to do so by building on top of an existing GUI library REF, and thus avoid the majority of the burden of developing a GUI library themselves REF.

Chapter 2

Existing Issues

2.1 Object Oriented Concepts

wxHaskell is built on top of an existing GUI library called wxWidgets. However wxHaskell is a GUI library for Haskell and wxWidgets is a GUI library for C++, and Haskell and C++ are very different languages, Haskell is a functional programming language and C++ is an object oriented language. Unfortunately wxHaskell exposes the object oriented concept of inheritance to the programmer and wxHaskell code is typically written using about twenty percent low level bindings to wxWidgets.

In wxWidgets inheritance is used to describe the type of many components. For example a button in wxWidgets has a type `wxButton` but it has many layers of inheritance as you can see in the image below. Because wxHaskell is a wrapper around wxWidgets some concepts from wxWidgets appear in wxHaskell, in the case of a button its type in wxHaskell is `Window (CControl (CButton ()))` which encodes some of the inheritance relationship.

wxHaskell consists of four key libraries, only two of which are typically used by a wxHaskell programmer. The lesser used of these is wxcore which is a set of low-level Haskell bindings to wxc where wxc is a C language binding for wxWidgets. The more used is wx which is a set of higher-level wrappers over wxcore. Most wxHaskell software is about eighty percent wx and twenty percent wxcore.

We have described how wxHaskell exposes object oriented concepts of the wxWidgets library which it wraps, both through encoding inheritance and the low-level bindings of wxcore. The reason all of this is unfavourable is because as programmers we have some choice in the languages we use, and a functional language like Haskell is chosen because it makes it easier to produce flexible, maintainable, high-quality software REF. Re-introducing concepts from languages that were not chosen is a compromise. Analogously if you were an object oriented programmer and were told that you are now only allowed to use sequential statements you probably would not be too happy.

While we can argue the merits of functional programming it is worth noting that Haskell and C++ are two solutions to different problems, they each solve their share of problems equally well. Haskell provides a high level of abstraction and few runtime errors while C++ provides fast execution time and a lot of library support. However if you have chosen a language to work with you should be able to stay within its constructs and paradigms.

2.2 Difficult to Install

Ease of downloading and installing libraries into sandboxes has become a staple of modern languages, with many modern languages like Rust, Swift and Elixir shipping with powerful package managers that automate this process. Haskell has made progress on this front with the package manager Stack.

Before Stack existed it was not uncommon to be stuck with dependency conflicts between libraries that your project depends on. Dependency conflicts occur when libraries have conflicting version bounds on some mutually required library. For example if library-a requires library-c > 0.7 but library-b requires library-c < 0.7 then we have a dependency conflict since no version of library-c can satisfy both conditions. You might end up changing the version of library-a to a previous version that requires library-c 0.6 which then satisfies both conditions, however, now library-a also requires library-d 0.5 but library-c requires library-d > 0.6. This endless cycle of fixing dependency conflicts is commonly referred to as Cabal hell.

The Haskell tool Stack solves Cabal hell by providing sets of libraries which are guaranteed to work together without dependency conflicts. These sets of libraries are called resolvers and every week on Sunday night a new stable resolver is released. Using Stack we can easily add a dependency to a Haskell project by simply listing it in the project's dependencies. The next time the project is built using Stack the new dependency will automatically be downloaded and built to a location in a sandbox designated for the project.

wxHaskell is not in the current Stack resolver (we commonly just say ”in Stack”). This means if we want to build our project with the tool Stack, then wxHaskell has to be listed as an additional dependency, and there are no guarantees of avoiding conflicts with wxHaskell's dependencies. At the beginning of this final year project U·(TP)² was not building with Stack at all but rather had to built by directly invoking the GHC compiler. Andrew Butterfield later succeeded in getting the project building with Stack, the significance of which is reflected in the relevant commit message:

> UTP2 NOW BUILDS WITH stack ON OS X 10.11.16 !!!!

It is worth noting two things here. One is that the difficulty of getting U·(TP)² to build with Stack was because of dependencies like wxHaskell which are not in Stack and caused dependency conflicts. The second is that there are benefits to Stack apart from its resolvers, including isolated and reproducible builds, and an easy to use command line interface.

However installing wxHaskell is not *just* a matter of resolving dependency conflicts. We also need to install the C++ library wxWidgets which wxHaskell is a wrapper around. The instructions for installing wxWidgets are different per platform due to their not being a well-established C++ package manager. Furthermore, on macOS, installing wxWidgets requires an install of the application XCode which on my machine weighs in at 10.46GB.

2.3 Difficult to Package

A goal of Andrew Butterfield's while developing U·(TP)² was to reach a point where operating system native applications of U·(TP)² could be distributed e.g. `.deb` packages for Debian or `.app` bundles for macOS, or if not native applications then at least executables. This proved difficult for the existing project as it was not being successfully built on macOS and was difficult to build on Linux, however executables for Windows do exist and are hosted on the project's homepage. At least on macOS the difficulties in building the project are largely related to wxHaskell, for reasons discussed in the previous section 2.2.

> Students at TCD have successfully built it on Linux (Ubuntu). It should run in principle on Max OS X as well, but I have not been able to get this to work (help would be appreciated).
>
> – scss.tcd.ie/Andrew.Butterfield/Saoithin

2.4 Conclusion

In respect of the object oriented concepts exposed by the wxHaskell library, and the difficulty in building U·(TP)² and creating operating system native applications of U·(TP)² – in both of which wxHaskell plays a role – we decided to attempt building a GUI for U·(TP)² using an alternative GUI library, one we hoped would alleviate all of the problems associated with wxHaskell.

Chapter 3

A New Hope

3.1 Haskell GUI Libraries

Unfortunately the state of GUI programming in Haskell is not in a great place. There do exist many GUI libraries but they tend to fall into one of two categories. Some provide direct access to GUI facilities through bindings to an imperative library, wxHaskell falls into this category. Most of the more powerful GUI libraries fall into this category, because they can leverage the existing power of the imperative language they provide a binding to. Others present more high-level programming interfaces, and have a more declarative, functional feel. These libraries tend to not provide GUI support directly but rely on a library like wxHaskell to provide the necessary GUI bindings.

> There is a large number of GUI libraries for Haskell. Unfortunately there is no standard one and all are more or less incomplete. In general, low-level veneers are going well, but they are low level. High-level abstractions are pretty experimental. There is a need for a supported medium-level GUI library.
>
> – wiki.haskell.org/Applications$_{\text{andlibraries}}$/GUI$_{\text{libraries}}$

3.2 Threepenny-gui

Threepenny-gui is a GUI library for Haskell which falls into the previously mentioned second category, it provides high-level abstractions with a declarative, functional feel. However it does not rely on another library like wxHaskell to provide GUI bindings, Threepenny-gui is a stand-alone GUI library. As a stand-alone GUI library Threepenny-gui does not rely on any non-Haskell dependencies, in stark contrast with wxHaskell.

How does Threepenny-gui display things on-screen? Threepenny-gui does not create bindings to any system calls to display a GUI, this means that Threepenny-gui applications are not operating system native applications. Threepenny-gui's key distinguishing factor is that it uses the web browser as a display. Web pages like docs.google.com are examples of powerful web applications, applications that use the web browser to display a GUI. There are many powerful web applications that provide an experience that is not compromised because the application was written as a web application instead of as an operating system native application. A notable part of the experience when using a web application like Google Docs is that an installation is not required, a web browser which is the necessary software to display the GUI, is something which most people already have installed. Threepenny-gui manages to avoid relying on another Haskell library for GUI bindings, and manages to avoid any non-Haskell dependencies. It does so by requiring a piece of software to display a GUI that most people already have installed, a web browser.

Because Threepenny-gui manages to avoid GUI related dependencies, by using the web browser as a display, the pain of installing these dependencies is removed and installing Threepenny-gui is easy. At the time Threepenny-gui was chosen it was not in Stack, however only one of its dependencies was not in a Stack. Once a library's entire dependencies are in Stack it is trivial to get that library in Stack. A few weeks after discovering Threepenny-gui it was in the latest Stack resolver.

Because Threepenny-gui uses the web browser as a display, this means that what is being rendered to the user is ultimately just HTML and CSS. How Threepenny-gui works is that it provides functions to write and manipulate HTML, it also allows the programmer to load CSS files and to run JavaScript. How Threepenny-gui works will be explained in more detail later on but in essence it is a wrapper around the languages of modern web development, this means the full power of modern development can be leveraged in a Threepenny-gui application. Another benefit of Threepenny-gui being a wrapper around HTML, CSS and JavaScript is that if you are familiar with these web development technologies then Threepenny-gui has a relatively gentle learning curve compared to other Haskell GUI libraries.

We have mentioned that Threepenny-gui provides high-level abstractions, with a declarative, functional feel. This is largely due a concept called Functional Reactive Programming (FRP) which is at the heart of Threepenny-gui. FRP will be explained in more detail later on, for now it is sufficient to know that FRP is a style of programming which is very much in line with the functional programming ideology, of declarative high-level semantics. Heinrich Apfelmus is the author of a popular FRP library for Haskell named reactive-banana. Apfelmus created Threepenny-gui to explore the application of FRP to building a GUI.

3.3 Threepenny-gui for U·(TP)²

Threepenny-gui was chosen for U·(TP)² because of the above reasons. It is easy to install, in stark contrast to wxHaskell. It has a gentle learning curve if you are already familiar with web development technologies. Finally, the strong focus on FRP within Threepenny-gui promotes writing a GUI in a declarative manner, in a style in-line with the functional programming ideology.

While Threepenny-gui has these many benefits it is still a young library and would likely have some flaws, which would later be confirmed. Threepenny-gui was only started in July 2013 and at the current time of writing is on version 0.7.1. However, for a functioning GUI library Threepenny-gui has quite a small code base which makes it easier to get involved and find solutions to these flaws. The small code base also means that Threepenny-gui is very maintainable which is vital for its longevity. Part of the reason for the small code base is the fact that Threepenny-gui leverages the power of existing web development technologies, letting these existing and widely prevalent technologies do the heavy lifting.

Chapter 4

Threepenny-gui

4.1 Introduction

As the project progressed flaws of Threepenny-gui were discovered and addressed. This required making modifications to Threepenny-gui's source code. In light of this it is beneficial to have a deeper understanding of how Threepenny-gui operates, which will make understanding Threepenny-gui's flaws and how they were addressed much easier later on. This chapter provides an overview of how Threepenny-gui operates and then provides an in-depth walk-through of a small Threepenny-gui application.

4.2 Overview

Threepenny-gui uses the web browser as a display. This means that a user views a Threepenny-gui application in their browser, and what is rendered in their browser is HTML and CSS, which can be manipulated by JavaScript. To solidify this idea that a Threepenny-gui application is ultimately HTML and CSS the screenshot below shows a simple Threepenny-gui application being displayed in a web browser. The web browser's developer tools are open to show the HTML structure of the application, which can be seen on the right.

The screenshot above shows how a Threepenny-gui application consists of HTML. However it only shows a static view of the application and applications generally need to be dynamic; the displayed HTML needs to be able to change in structure, in response to user input for example. These manipulations are done in the browser by JavaScript. Any Threepenny-gui code which manipulates displayed elements is converted from Haskell to JavaScript and evaluated in the web browser. For example we might want to append a list item `<li>` with text ”Ferrari” to a list `<ul>` of car names, and have written the appropriate Haskell code (below). At runtime this Haskell code is converted to JavaScript and evaluated in the browser.

```
UI.ul #+ [UI.li # set UI.text "Ferrari"]
```

Appending to a list in Threepenny-gui

So far we have covered the ideas that Threepenny-gui applications are displayed using HTML and CSS in a web browser, and that manipulations occur by converting Haskell code to JavaScript and evaluating it in the web browser. One important question is how a Threepenny-gui application knows when to apply the manipulations, when to evaluate the JavaScript? For example we might only want the colour of a HTML element to change when the user presses a specific button, in this case we are waiting for input from the user and once that input is received JavaScript is evaluated. Wherever our Threepenny-gui application is interested in a certain event, such as a user pressing a button, interest in that event is registered with the web browser which is displaying the application. Whenever the event occurs in the browser, the Threepenny-gui application is informed and may send additional JavaScript code to the browser to be evaluated.

4.3 Walkthrough

We now have an overview of how a Threepenny-gui application is displayed in the browser, including conversion to JavaScript code and how browser events such as button clicks are handled. We will now look at the life-cycle of a Threepenny-gui in more detail, by looking at a minimal working Threepenny-gui application. While working our way through the application we will be referring to the image below which describes the life-cycle of a Threepenny-gui application.

The Haskell code of the Threepenny-gui application we will walk-through is below. In particular we are concerned with the four lines of the body of the function `app`. The remaining code is necessary

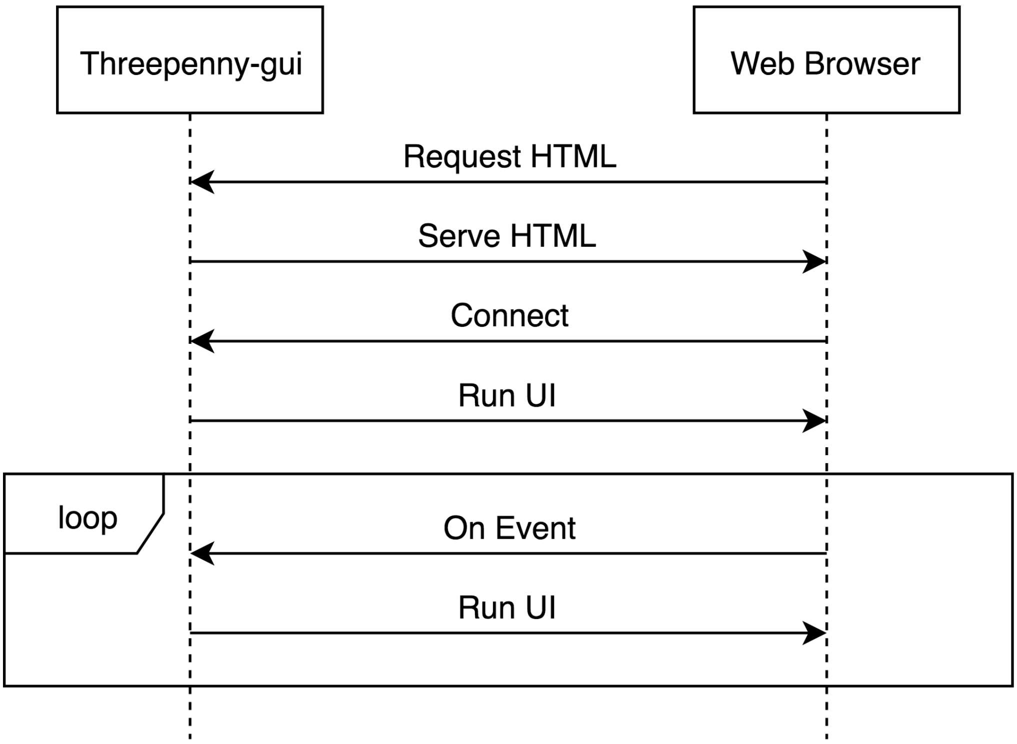

Figure 4.1: Life cycle of a Threepenny-gui application.

boilerplate to achieve a full working application. The first line of `app` creates a button with text "Click me!". In the second line we attach that button to the HTML `<body>`. The third line causes its body to be evaluated when a user clicks the button. The fourth line is what is then evaluated, it changes the button's text to "I have been clicked!".

```
module Main where

import qualified Graphics.UI.Threepenny      as UI
import           Graphics.UI.Threepenny.Core

main = startGUI defaultConfig app

app window = do
  button <- UI.button # set UI.text "Click me!"
  getBody window #+ [element button]
  on UI.click button $ \(x, y) ->
    element button # set UI.text "I have been clicked!"
```

We have described the application code at a high-level, now we will look in more detail at what occurs at runtime. When we execute the compiled code a local HTTP server is started, the server serves our Threepenny-gui application at the address `localhost:8000` by default. We can visit this address in our browser to view our Threepenny-gui application. When we visit `localhost:8000` in our browser a HTTP GET request is sent to the server and the server responds with an HTML file, this HTML doesn't yet contain any HTML describing our Threepenny-gui application. This HTTP GET request and the response correspond to the first two arrows in our life cycle diagram.

Included in this initial HTML file is some JavaScript which is evaluated in the browser, it opens a connection to the server. This is the third arrow in our life cycle diagram. The type of connection opened is called a WebSocket connection, which stays open until the user closes their browser tab. The benefit of maintaining an open connection between the server and the browser is that the server can send data to the browser whenever it wants to, this means the server can update what is being displayed at any time. For example we might want to set a button to a red colour after a timer expires. Because a WebSocket connection is open, the server can send JavaScript code to the browser when the timer expires, this JavaScript code is evaluated in the browser and sets the button to a red colour. To further see why maintaining an open connection is important we can consider the traditional alternative to a WebSocket. In a traditional web application the browser sends HTTP requests to the server and the server responds, the server can only send data to the browser in response to a browser's HTTP request. Considering our timer example, for the browser to know when the timer has expired the browser would have to be constantly polling the server.

Continuing with our example application, once the WebSocket connection has been opened our Threepenny-gui application code is evaluated, this corresponds with the fourth arrow in our life cycle diagram. In the second line of `app`, JavaScript code is sent from the server to the browser to be evaluated, this code adds the button element from the first line to the HTML `<body>`. In the third line the server tells the browser that it should be informed of any clicks on the button, in other words we are registering an event handler that is triggered by clicks to the button.

Finally we will consider the loop in the life cycle diagram. The browser informs the server whenever the button click event occurs, this corresponds to the fifth arrow in the life cycle diagram. When the server receives this information the fourth line of `app` is run, sending JavaScript code to the browser to change the button's text to "I have been clicked!" which corresponds to the final arrow of the life cycle diagram. This event loop will continue until either the user closes the browser tab or the server is killed.

Part II

Implementation

Chapter 5

A Right-Click Menu

5.1 Background

Right-click menus are widely used in the existing U·(TP)² application, below is a screenshot showing a right-click menu on the application's home screen. Building a custom right-click menu using Threepenny-gui represented, to some degree, an investigation into the feasibility of using Threepenny-gui to build an entire GUI for U·(TP)². This is both because a right-click menu is one of the more complex components of a GUI and also because of the widespread use of right-click menus in U·(TP)².

Threepenny-gui does not provide a facility to build a right-click menu. You might expect, that a GUI library would provide support for building a right-click menu, since it seems like one of the fundamental parts of a GUI. However Threepenny-gui's approach is different to a traditional GUI library, it acts as a wrapper around existing web technologies, leveraging their power. This means that the problem of building a right-click menu in Threepenny-gui instead becomes the problem of building a right-click menu using web technologies.

Building a right-click menu using web technologies is not entirely straightforward either. There exists a HTML specification for building a right-click menu however it is only enabled by default by Mozilla's Firefox browser. Google's Chrome browser and Apple's Safari have implemented the specification however is must be enabled via a developer flag. Microsoft's Edge does not support the specification. This HTML specification for building a right-click menu only existed as a recommendation by Mozilla at the time of my investigation though it was accepted to the HTML Living Standard on January 17 2017, however browser support is as previously mentioned.

5.2 Implementation

While most major browser's do not, at least by default, support right-click menus based on the HTML specification, all major browsers support the JavaScript `contextmenu` event which can be used to build a right-click menu, albeit with a bit more work. JavaScript events, in particular the `contextmenu` event and how it can be used to build a right-click menu is explained below.

HTML consists of a tree of elements such as `<body>`, `<div>` or `<button>`, an example of HTML's tree structure is shown in the figure below. When a JavaScript event occurs at one of these elements it propagates upward through the tree of elements; downward propagation is also possible, though upward propagation is most common. For example when a user clicks on an element a `click` event is fired at that element and propagates upward through the tree of elements. JavaScript event handlers can be bound to elements, such that when an event propagates through an element it can trigger an event handler. This idea of event propagation and handling is very similar to the idea of exception propagation and capturing which is available in most programming languages.

According to Mozilla's documentation "The `contextmenu` event is fired when the right button of the mouse is clicked (before the context menu is displayed), or when the context menu key is pressed". This simply means that the `contextmenu` event is fired when a user right-clicks, the context menu key mentioned refers to the fact that a user can simulate a right-click on some keyboards. An event handler for a `contextmenu` event is thus a function that will only be evaluated when a user right-clicks.

To build a right-click menu we need to know two important things, when a user right-clicks on an element and the coordinates of the right-click. If we know when a user has right-clicked on an element then we know when to display our right-click menu, if we know the coordinates of the right-click then we know where to display our right-click menu.

To solidify our goals: we want to display a custom right-click menu R when a user right-clicks on a element E. Our approach to building this right-click menu is to write an event handler that is triggered by a `contextmenu` event fired by the element E. When this event handler is evaluated we will display a custom right-click at the coordinates given in the `contextmenu` event. The right-click menu we will display will simply be built from standard HTML elements such as `<div>`, with some styling.

We previously discussed writing an event handler in the background section on Threepenny-gui, the relevant code is shown again below, here the event handler created would be triggered by a `click` event fired by the `button` element. To build a right-click menu we want to accomplish something similar but our event handler needs to be triggered by a `contextmenu` event instead of a `click` event. The problem was, at the time, Threepenny-gui did not provide a `UI.contextmenu` function similar to `UI.click`.

```
on UI.click button $ \(x, y) ->
  -- event handler body
```

A pull request is a request to merge code with an existing code base. We sent a pull request to the Threepenny-gui repository which added a `UI.contextmenu` function to Threepenny-gui, the pull request was accepted and the code is now part of Threepenny-gui. Now with `UI.contextmenu` it is possible to create event handlers that are evaluated when a user right-clicks an element.

Now that Threepenny-gui supports writing event handlers for `contextmenu` events the next step is to write a library which leverages that capability and allows a user to build right-click menus. We built a library called threepenny-gui-contextmenu which is publicly available and provides this functionality. The README of threepenny-gui-contextmenu is available as an appendix.

A custom right-click menu built with threepenny-gui-contextmenu, some items in the right-click menu change the button's colour, other items open a nested menu.

A custom right-click menu built for U·(TP)² using threepenny-gui-contextmenu.

5.3 Feasibility

Implementing threepenny-gui-contextmenu was not straightforward, even after `UI.contextmenu` had been added to Threepenny-gui. Conditions had to be taken into account which were not initially considered, for example when our threepenny-gui-contextmenu event handler is triggered on a right-click, we need to prevent the `contextmenu` event from propagating further, otherwise the standard browser right-click menu would also be shown in addition to our custom right-click menu. Another difficult case when a user's mouse leaves a right-click menu, all nested menus are closed but the root menu remains open, as depicted below.

We mentioned at the beginning of this chapter that building a custom right-click menu using Threepenny-gui represented, to some degree, an investigation into the feasibility of using Threepenny-gui to build an entire GUI for U·(TP)². Considering the difficulties in doing so, it raises the question of whether Threepenny-gui is a feasible choice for building a GUI for U·(TP)²? Our answer is that it is. This answer is justified as follows. While implementing a right-click menu was difficult, it also was possible, this serves as an indicator that we can use Threepenny-gui as an alternative to wxHaskell. More importantly however, while implementing a right-click menu we managed to contribute to Threepenny-gui's source code. Considering the poor state of the Haskell GUI space, the fact that we can contribute to a library like Threepenny-gui, and ever so slightly improve the state of the Haskell GUI space, is a large positive.

Chapter 6

Layout

6.1 Background

What is displayed in a GUI is, at a high-level of abstraction, simply a set of elements in a certain layout. For example a GUI might consist of a navigation bar above a main viewing area, a simple two element layout. Each of these two elements might again consist of a layout of further elements, for example the navigation bar might consist of multiple tabs in a horizontal layout. Layout is simply an unavoidable consideration when building a GUI.

HTML and CSS are powerful tools which allow us to create complex layouts, however the means to do so can also be complex. Threepenny-gui leverages the power of these web technologies meaning that any layout in HTML and CSS is also possible in Threepenny-gui. According to the Threepenny-gui homepage ”This is a blessing, but it can also be a curse, so the library includes a few layout combinators to quickly create user interfaces without the need to deal with the mess that is CSS”.

The layout combinators (functions) that Threepenny-gui provide allow us to layout elements in tables, where each element is contained in a cell of the table. These tables are displayed in the browser using `<table>`, `<tr>` and `<td>` HTML elements. HTML tables have long been the de facto standard for writing layouts in HTML documents. However they have limitations; in particular HTML table layouts are not responsive, elements have a static size that will not change based on screen size. Heinrich Apfelmus acknowledges their limitation, stating that they ”tend to behave unpredictable, especially when content size changes dynamically".

6.2 Flexbox

Flexible box, or Flexbox, is a CSS specification which was introduced as a recommendation by W3C in May 2016, and is currently supported by all major browsers. Flexbox provides a means for writing responsive layouts, Flexbox allows elements to grow to fill available space, or to shrink to avoid overflow. We can also do more complex things like specify that elements should have sizes according to a certain ratio, or have elements wrap onto new lines if there is not enough space on the current line.

> In the flex layout model, the children of a flex container can be laid out in any direction, and can "flex" their sizes, either growing to fill unused space or shrinking to avoid overflowing the parent. Both horizontal and vertical alignment of the children can be easily manipulated.
>
> – Flexible Box Layout, W3C Candidate Recommendation, 26 May 2016

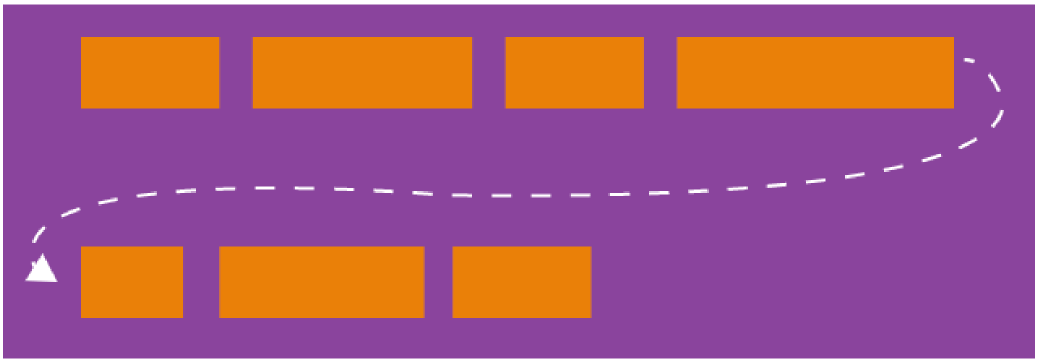

Figure 6.1: Using Flexbox to wrap elements onto a newline on overflow.

6.3 Implementation

Heinrich Apfelmus appears in favour of Flexbox, writing that Flexbox "apperas to solve most of the layout woes. Flexboxes may be a good start for implementing proper layout combinators in

Haskell". Flexbox would allow us to write responsive layouts for U·(TP)^2 and it was a direction for Threepenny-gui that Apfelmus was in favour of. For these reasons we decided to write a library that would add Flexbox support to Threepenny-gui.

Flexbox is a CSS specification, this means Flexbox layouts are written using CSS properties, also known as CSS rules. To write Flexbox layouts, it is simply a matter of applying the correct CSS properties to a parent element and its children elements. Below is some example HTML showing how we can achieve three elements in a ratio of 1:2:1 using CSS Flexbox properties, the graphical result of this HTML is also shown below. Note that some additional styling code is not shown.

```
<div style="display: flex;">
  <div style="flex-grow: 1;">foo</div>
  <div style="flex-grow: 2;">foo</div>
  <div style="flex-grow: 1;">foo</div>
</div>
```

Listing 1: HTML for **??**.

Figure 6.2: Three elements in ratio.

We wrote a library called threepenny-gui-flexbox which provides a method of writing the necessary CSS properties for Flexbox and applying them to Threepenny elements. This library is in Stack and its README is attached as an appendix. The code

threepenny-gui-flexbox instead of HTML is shown below, again note that some additional styling code is not shown.

```
flex_p UI.div [
       (foo, flexGrow 1)
     , (foo, flexGrow 2)
     , (foo, flexGrow 1)
     ]
where foo = UI.div # set UI.text "foo"
```

Listing 2: Threepenny-gui for

Chapter 7

File Selection

7.1 Browser Security

Chapter 8

Electron

Chapter 9

Electron Packager

Chapter 10

Directory Selection

Chapter 11

Functional Reactive Programming

Chapter 12

Abstract GUI Layer

Chapter 13

Conflicting Architectures

Chapter 14

Web Development Libraries

Part III

Reflections

Chapter 15

Conclusion

15.1 Edit-Compile-Evaluate

15.2 Threepenny-gui's Future

Part IV

Appendices

Chapter 16

Terminology

Chapter 17

threepenny-gui-contextmenu

Chapter 18

threepenny-gui-flexbox

Bibliography

[Jr2006] Jr, N. (2006). My article.