Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

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

I would like to thank Andrew Butterfield for supervising this project and in particular for giving me the opportunity to get involved with the Threepenny-gui project. I would also like to thank Heinrich Apfelmus, the maintainer of Threepenny-gui, for taking the time to respond to my queries.

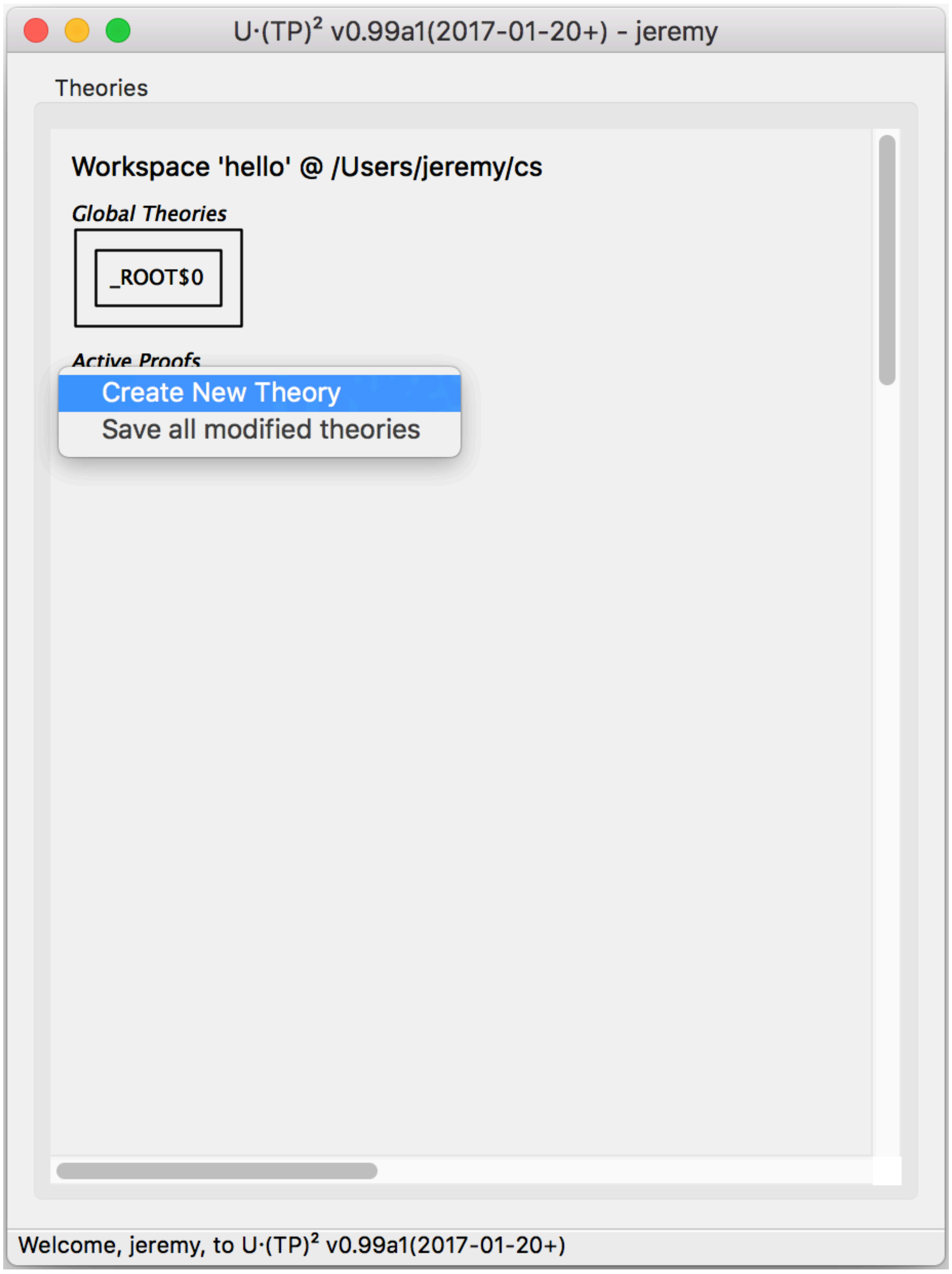
Contents

Background

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 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 as early as July 2003, [**?**]. Though wx-Haskell’s official history only begins in January 2007 when the project was taken over by a new set of maintainers, [**?**]. The goal of wxhaskell is to provide an industrial strength GUI library for Haskell, the wxHaskell team attempt to do so by building on top of an existing GUI library, and thus avoid the majority of the burden of developing a GUI library themselves, [**?**].



U·(TP)^2 v0.99a1(2017-01-20+) - jeremy

Workspace 'hello' @ /Users/jeremy/cs

Welcome, jeremy, to U·(TP)^2 v0.99a1(2017-01-20+)

Figure 1: Existing U·(TP)^2 home screen.

Existing Issues

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 provides low-level Haskell bindings to 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. This is unfavourable because as programmers we have some choice in the languages we use, we choose a language because of features that appeal to us. We might choose a functional language like Haskell because features of the language such as lazy evaluation and higher-order functions allow us to write highly modular programs, programs that are much smaller and easier to write than conventional ones, [**?**]. It is a compromise to be exposed to features from a language that we did not choose.

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.

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 Haskell 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 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.

Difficult to Package

A goal of Andrew Butterfield's while developing U·(TP)^2 was to reach a point where standalone 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

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.

A New Hope

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, others present more high-level programming interfaces and have a more declarative, functional feel, [?]. wxHaskell falls into the first category, of bindings to an imperative library. Most of the more powerful GUI libraries fall into this category, because they can leverage the power of the imperative language they provide a binding to. Libraries in the second category, high-level 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.
>
> – [?]

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 standalone 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 standalone 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.

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.

Threepenny-gui

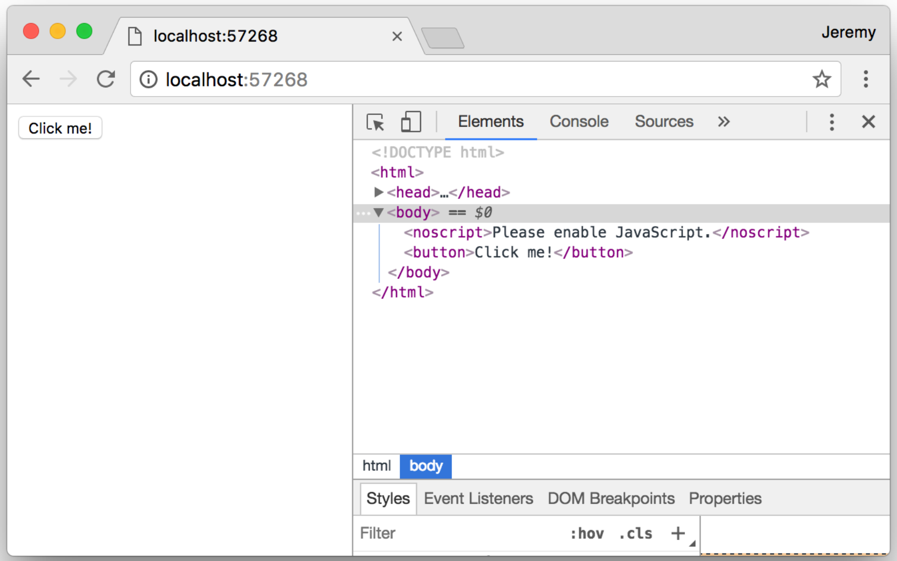
Introduction

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.

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 the idea that a Threepenny-gui application is ultimately HTML, Figure 2 shows a simple Threepenny-gui application being displayed in a browser. The browser's developer tools are open, showing the HTML structure of the application.

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



```
<!DOCTYPE html>
<html>
▶<head>…</head>
▼<body> == $0
    <noscript>Please enable JavaScript.</noscript>
    <button>Click me!</button>
  </body>
</html>
```

Figure 2: A Threepenny-gui application is ultimately HTML.

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"]
```

Listing 1: 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.

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 Figure 3 below which describes the life-cycle of a Threepenny-gui application.

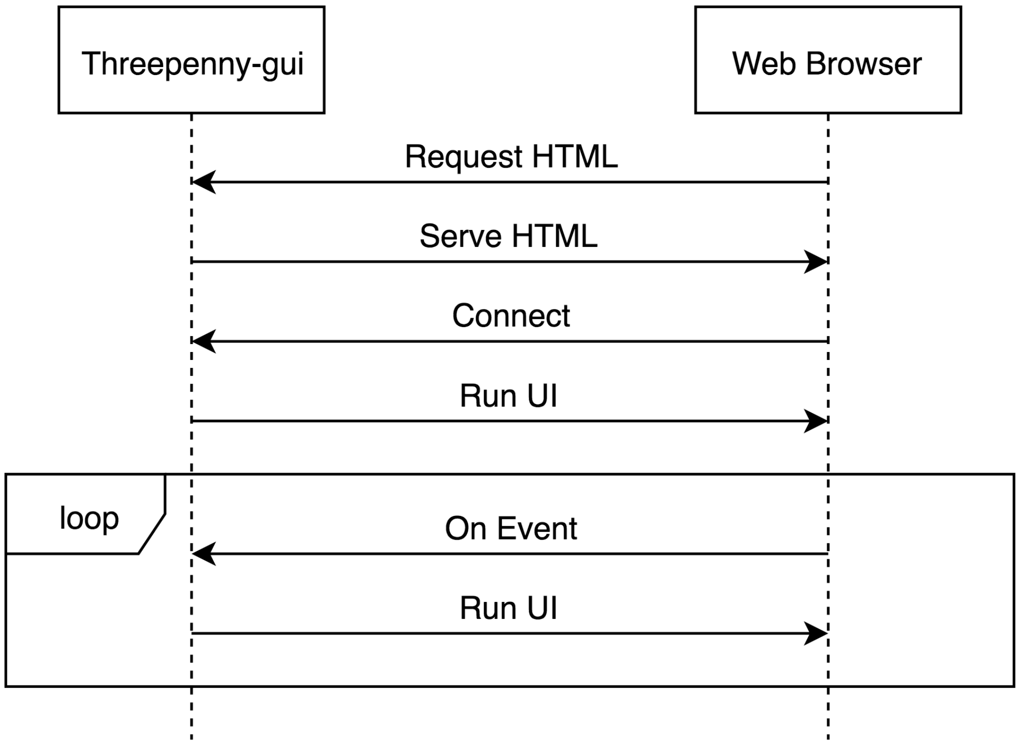


Figure 3: Life cycle of a Threepenny-gui application.

The Haskell code of the Threepenny-gui application we will walk-through is in Listing 2 below. 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 evaluated when a user clicks the button, changing the button’s text to ”I have been

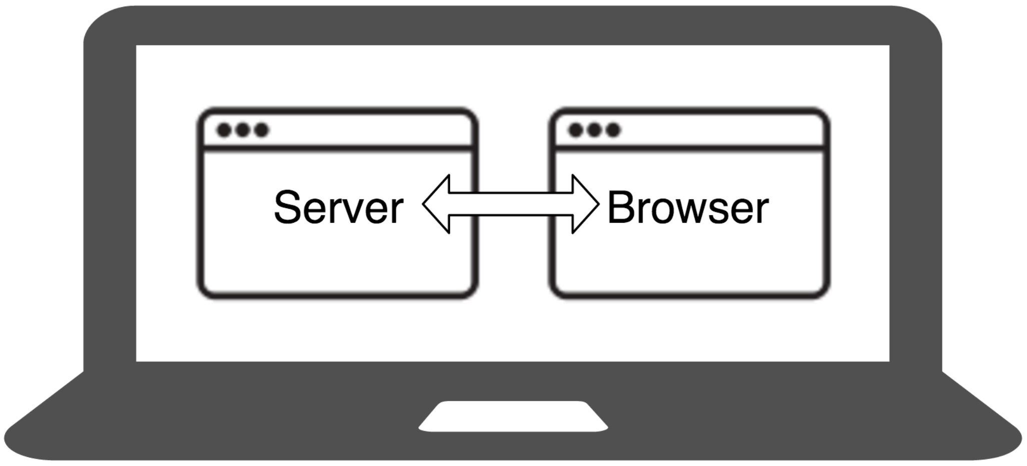


Figure 4: Threepenny-gui applications are served by a local server.

clicked!".

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

Listing 2: A minimal Threepenny-gui application.

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.

Implementation

A Right-Click Menu

Background

Right-click menus are widely used in the existing U·(TP)^2 application, Figure 1 shows an example of 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 is more of a 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 at the time of writing 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, and Microsoft's Edge does not support the specification.

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>`, `<p>` or `<button>`, an example of HTML's tree structure is shown in Figure 5. 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.

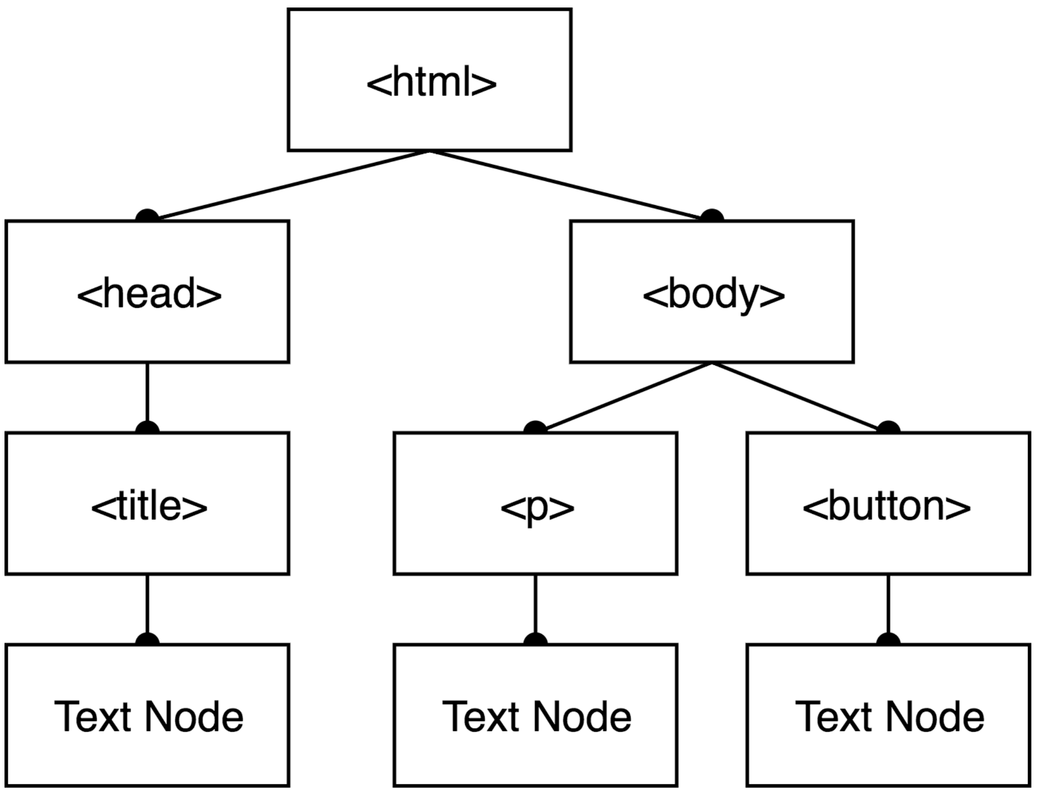


Figure 5: Tree structure of a HTML document.

We previously discussed writing an event handler in the background section on Threepenny-gui, the relevant code is shown again below in Figure 3, 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
```

Listing 3: Registering an event handler for clicks on a button.

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 included as an appendix.

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 shown below in Figure 7.

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

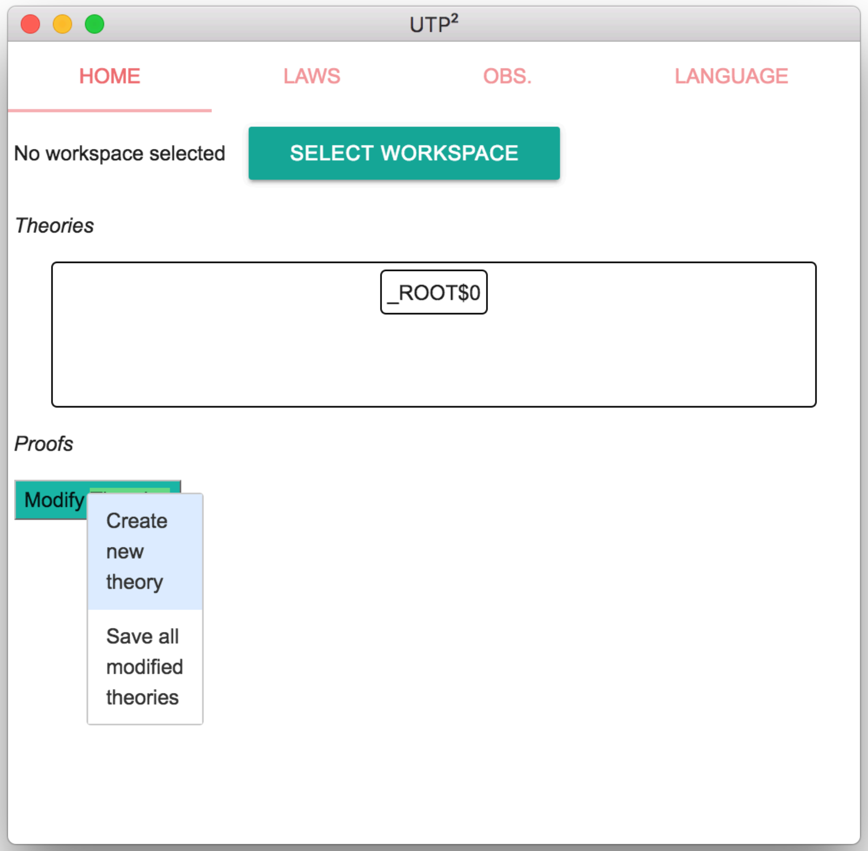


Figure 6: A right-click menu in U·(TP)² using Threepenny-gui.

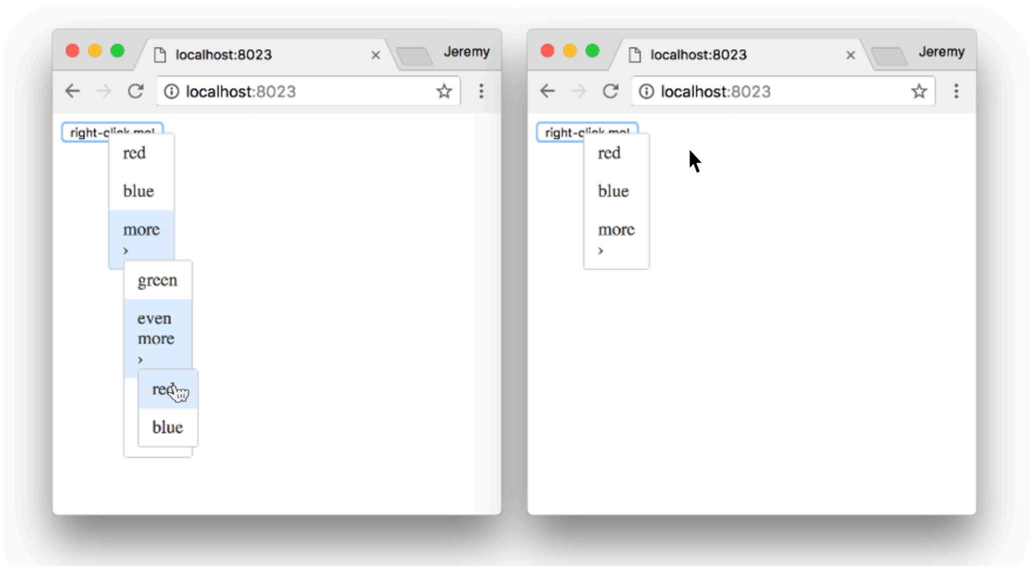


Figure 7: Leaving a nested right-click menu.

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 and that 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.

Layout

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 which is possible using HTML and CSS is also possible in Threepenny-gui. While HTML and CSS are powerful tools they can also be confusing, especially for those who are only looking for a GUI library in Haskell and are unfamiliar with HTML and CSS.

> You have all capabilities of HTML at your disposal when creating user interfaces. 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”.

Flexbox

Flexbox, is a CSS specification for writing responsive layouts, it 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.
>
> – [**?**]

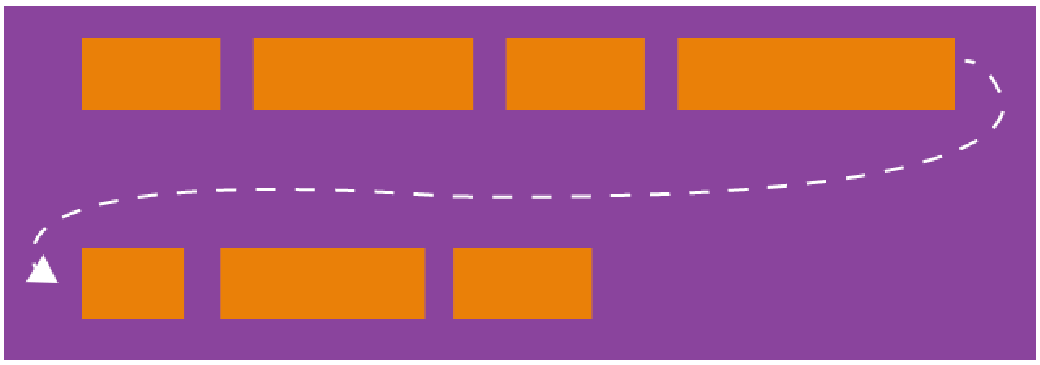


Figure 8: Using Flexbox to wrap elements onto a newline.

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”, [**?**]. Because Flexbox would allow us to write responsive layouts for U·(TP)^2 and because it is a direction for Threepenny-gui that Apfelmus in in favour of, we decide to write a library to add Flexbox support to Threepenny-gui.

Flexbox is a CSS specification, this means Flexbox layouts are written using CSS properties. To write Flexbox layouts, it is simply a matter of applying the correct CSS properties to a parent element and its children elements. Figure 9 shows three elements in a ratio of 1:2:1; this is a responsive layout written with Flexbox, meaning that the ratio of the elements will be maintained on different screen sizes. The HTML code with the necessary CSS properties to achieve Figure 9 is shown in listing 4, note that some additional styling code is not shown.

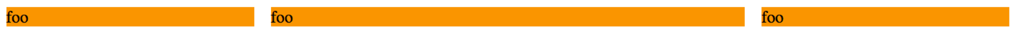


Figure 9: Three elements in ratio 1:2:1

```html
<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 4: HTML code for Figure 9

We published a library called threepenny-gui-flexbox which is in Stack. threepenny-gui-flexbox provides a method of writing Flexbox CSS properties and converting them to the format expected by Threepenny-gui, also included in the library are functions which provide abstractions for common patterns. For more detail threepenny-gui-flexbox's README is attached as an appendix. The code to achieve Figure 9 using Threepenny-gui and threepenny-gui-flexbox instead of HTML is shown in listing 5, again note that some additional styling code is not shown.

```haskell
UI.div # setFlex parentProps #+ [
    (UI.div # set UI.text "foo" # setFlex (flexGrow 1))
  , (UI.div # set UI.text "foo" # setFlex (flexGrow 2))
  , (UI.div # set UI.text "foo" # setFlex (flexGrow 1))
  ]
```

Listing 5: Threepenny-gui code for Figure 9

File Selection

Background

When a user runs the existing U·(TP)^2 application for the first time the first window presented to the user is a file selection dialog. The dialog asks the user to select a directory, which will be the application's workspace. The workspace is a directory which contains files that persist application state. Because of their existing use in U·(TP)^2, it is necessary to be able to implement directory selection dialogs with Threepenny-gui.

Threepenny-gui does not itself present any facilities for implementing directory selection. This again, similar to a right-click menu, seems like fundamental functionality that a GUI library should provide. In fact to implement directory selection, we must take a similar approach as we did with building a right-click menu. Because Threepenny-gui is a wrapper around existing web technologies, the problem of implementing directory selection with Threepenny-gui is instead a problem of implementing directory selection using web technologies.

Directory selection and file selection are very similar as far as a user is concerned, in both cases the user is presented with a window like the one in Figure 11, the only difference is the limitation of what the user is allowed to select. In HTML, the code for a directory selector and file selector are very similar, in fact a directory selector is simply a file selector with one additional attribute, see listings 6 and 7. For this reason we will first attempt to build a file selector in Threepenny-gui.

```
<input type="file">
```

Listing 6: A file selector in HTML.

```
<input type="file" webkitdirectory>
```

Listing 7: A directory selector in HTML.

Solution

We can quite easily write a Threepenny-gui application that results in a file selector as in listing 6, and prints the path of a selected file to stdout. Unfortunately, when running the application this will not print the file path we expect to stdout. When a user runs the application and selects a file such as `/Users/foo/bar.txt` the file path `C:\fakepath\bar.txt` is printed to stdout.

The reason that `C:\fakepath\foo.txt` is printed to stdout is because of a security feature that is present in all major browsers. If we are browsing a webpage, are prompted to select a file, and select a file, the server will only receive the file contents and the file name. The file path is obfuscated to appear as `C:\fakepath\<name>` where `<name>` is the file name. The reason for not revealing the full file path is so that the server cannot learn about the file system structure of a user. For example if the server were to receive a file path such as `/private/foo/bar.txt` then the server is aware of the existence of the directories `/private` and `/private/foo` on the user's file system, information the user might not have intended to share.

A local server is a server on a user's own machine, while a remote server is located on another machine. When browsing a webpage served by a remote server the browser security feature that obfuscates file paths makes sense, it is a security concern to be sharing details of our filesystem with a remote server. Recalling from the background chapter on Threepenny-gui, a Threepenny-gui application uses a local server to serve the application as a webpage, see Figure [[][and [[]]. In this case of browsing a webpage served by a local server the browser security feature does not make sense. We do not want to hide file paths from our own application.

In order for a Threepenny-gui application to receive the correct file path, the user needs to view the application in a browser which does not obfuscate the file path. We can solve this by shipping a browser as part of our Threepenny-gui application which has this security feature removed, this solution is discussed in the next chapter.

Electron

Background

Electron is a framework for creating standalone applications with web technologies. To display applications, Electron uses a modified version of the Chromium browser. Of particular interest, Electron's modified browser removes many security features found in most browsers. Included in the removed security features is file path obfuscation. This means that when a user is browsing a webpage through Electron's browser and selects a file, the server serving the webpage will receive the correct file path and not something of the form `C:/fakepath/<name>`.

Our goal is to integrate Electron with our Threepenny-gui application, so Electron's browser displays the application. After integrating Electron we can correctly implement file selection because Electron's browser does not obfuscate file paths, allowing the Threepenny-gui server to receive correct file path. Another benefit of displaying Threepenny-gui applications with Electron's browser is a consistent user experience. The reason for this is that all users would be viewing our Threepenny-guiapplication using Electron's browser; instead of their own installed browser, which may be different for each user. Browser's have different levels of support for web standards which results in a inconsistent user experience. Table 1 shows scores of different browsers for their support of the HTML 5 specification.

Using Electron to provide standalone applications is an open issue on the Threepenny-gui repository, issue `#111`. There are three chronological steps to issue `#111`. The first step is using Electron to display a Threepenny-gui application. The second step is being able to package the Threepenny-gui application as a standalone application so that it can be easily distributed without having to compile

Table 1: Browser scores for support of HTML5, from html5test.com on 01-05-2017.

Browser	Score
Chrome 57	519
Firefox 52	474
Edge 15	473
Safari 10.1	406

code or even touch the command line at all. Finally step three is to write a Haskell package to automate the first two steps.

Electron Integration

To display our Threepenny-gui application with Electron we cannot simply ask a user to download Electron's browser and view our Threepenny-gui application with it. This is because Electron's browser cannot be downloaded as a standalone application, instead Electron provides an API for managing browser windows. Electron provides its own JavaScript runtime which exposes this API. To open an Electron browser window we have to write a JavaScript script that includes a call to the Electron API that opens a browser window.

We wrote the necessary JavaScript script to display a Threepenny-gui application using Electron. The script executes the compiled Threepenny-gui application, starting the Threepenny-gui application's server. The script waits until the server is running then opens an Electron browser window with the URL pointing at the local server. Finally the script manages shutdown of the application, for example we have to consider the expected behaviour on macOS where clicking the red 'x' on an application's window only closes the window but leave the process running.

Electron Packager is a tool for packaging applications built with Electron into standalone applications. Now that we have our Threepenny-gui application being displayed using Electron we can use Electron Packager to create a standalone application for the current platform e.g. `.deb` packages for Debian or `.app` bundles for macOS. The application produced is entirely standalone, including all necessary dependencies such as the binaries of Electron and our Threepenny-gui application, which means it can be easily distributed.

We sent pull request #169 to the Threepenny-gui repository. The pull request includes the necessary script to display a Threepenny-gui application using Electron, a guide on how to integrate a Threepenny-gui with Electron using the script, and a link to a respository we setup which contains a minimal working example. After addressing initial feedback the pull request was accepted. The pull request addresses the first two steps of issue #111, it instructs users how to build and package Threepenny-gui applications with Electron but does not automate the process. The guide in the pull request is included as an appendix.

> Thanks a lot for this!
>
> – [?]

Directory Selection

Now that our Threepenny-gui application can be displayed using Electron, file selection works correctly, however we still need to accomplish directory selection. Revisiting 6 and 7 we can see that the difference is only a single attribute, namely `webkitdirectory`.

Setting attributes on a HTML element is done in Threepenny-gui by calling a specific function that Threepenny-gui exposes for each attribute, for example to set a `href` attribute we could use Threepenny-gui's `href` function. The problem is that Threepenny-gui does not provide such a function W for the `webkitdirectory` attribute, nor does Threepenny-gui expose the functions that allow us write to write W ourselves.

We had to fork the Threepenny-gui repository and expose the necessary functions that allow us to write W. The term ”fork” means a copy of a repository. With the fork of Threepenny-gui we can now write the function W to set `webkitdirectory` on a file selector, turning it into a directory selector. Figure 11 shows a directory selector in use in our Threepenny-gui implementation of U·(TP)^2, note that files are grayed out.

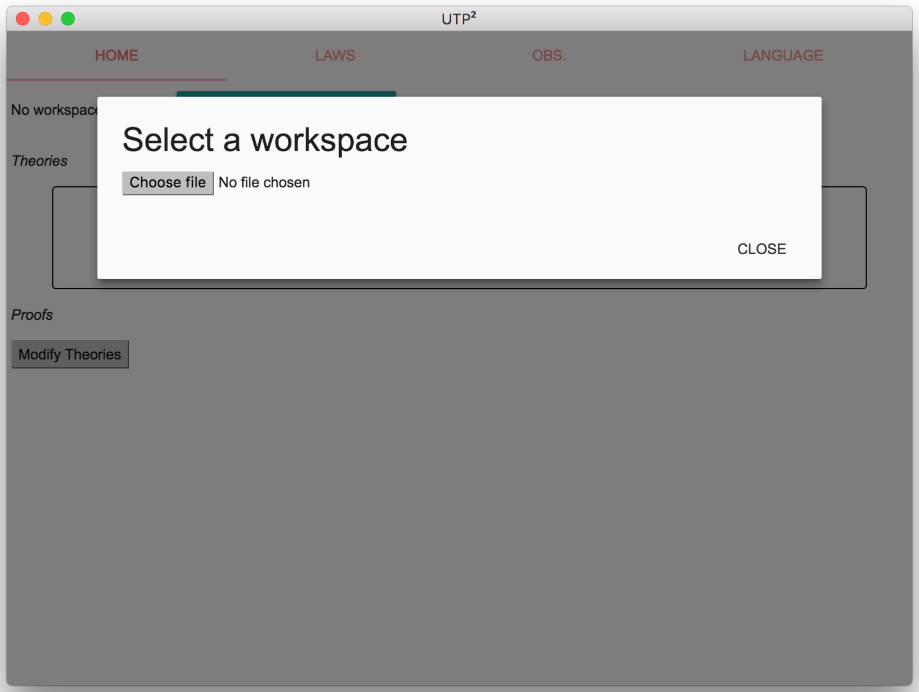
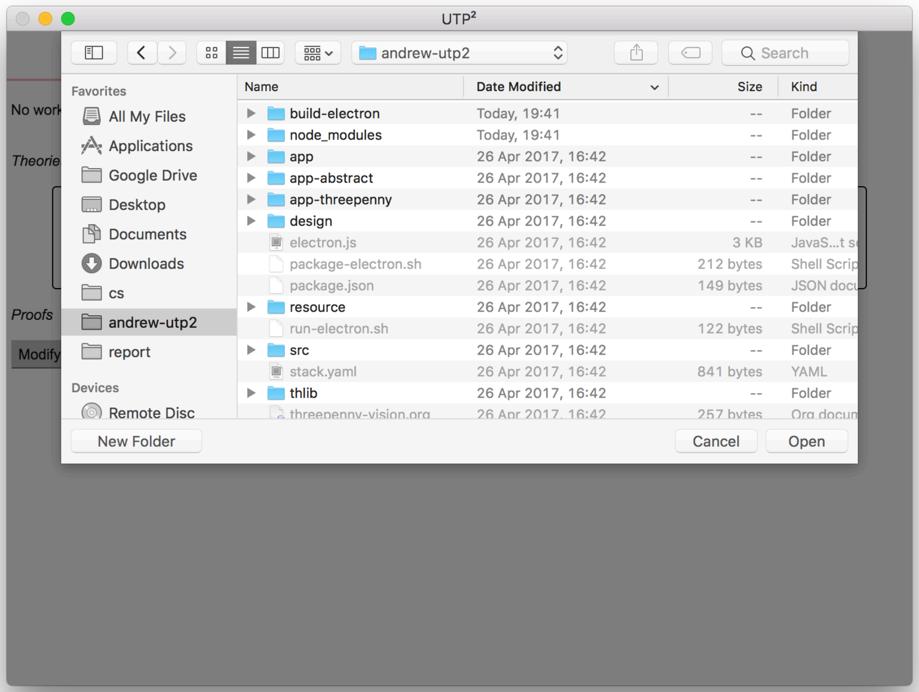


Figure 10: Workspace selection prompt in U·(TP)².



Name	Date Modified	Size	Kind
build-electron	Today, 19:41	--	Folder
node_modules	Today, 19:41	--	Folder
app	26 Apr 2017, 16:42	--	Folder
app-abstract	26 Apr 2017, 16:42	--	Folder
app-threepenny	26 Apr 2017, 16:42	--	Folder
design	26 Apr 2017, 16:42	--	Folder
electron.js	26 Apr 2017, 16:42	3 KB	JavaS...t s
package-electron.sh	26 Apr 2017, 16:42	212 bytes	Shell Scrip
package.json	26 Apr 2017, 16:42	149 bytes	JSON docu
resource	26 Apr 2017, 16:42	--	Folder
run-electron.sh	26 Apr 2017, 16:42	122 bytes	Shell Scrip
src	26 Apr 2017, 16:42	--	Folder
stack.yaml	26 Apr 2017, 16:42	841 bytes	YAML
thlib	26 Apr 2017, 16:42	--	Folder
threepenny-vision.org	26 Apr 2017, 16:42	257 bytes	Org docum

Figure 11: Workspace directory selection, follows on from Figure 10.

Functional Reactive Programming

Imperative or Reactive

Our Threepenny-gui application will in some places refer to data which will change over time, due to user input or for other reasons. For example, on the home screen an element displays the current workspace directory, this value is initially unset but receives a value once the user has selected a workspace. Updating elements to reflect changes in their dependant data can be done in an imperative style or reactive style.

```
text <- UI.div # set UI.text "No workspace set"
```

Listing 8: Element displaying the current workspace.

```
on UI.valuechange selector $ \newWorkspace ->
  -- User selected a workspace, need to update elements.
```

Listing 9: Handling when a user selects a workspace.

In the imperative style, when we write an element that depends on changing data e.g. Listing 8, we also write code to update that element wherever the data changes e.g. Listing 9. The code declaring the element and the code handling a change in the data may be in separate modules, or at least separate to some degree. This separation is a violation of the Law of Demeter, [?], which says we should minimize coupling between modules.

In the reactive style, when we write an element that depends on changing data we simply declare that it should have the *current* value of the data, this data is held in some variable D. Now the code handling a change in the data, namely Listing 9 simply updates D, any elements which depend on D will be updated automatically, see Figure 12. Notably the code handling a change to the data does not have to be concerned with elements depending on the data, we are no longer violating the Law of Demeter.

> A remote part of the program may change the [element], and this ”action at a distance” is not visible at the point where the counter is declared. In contrast, FRP specifies the whole dynamic behavior at the time of declaration
>
> – [?]

To summarise, functional reactive programming is preferable because when some data changes we don’t have to worry about updating the elements that depend on the data directly. Instead we update the elements indirectly, by emitting a new value of the data which the elements then receive. By not updating the elements directly we are not violating the Law of Demeter.

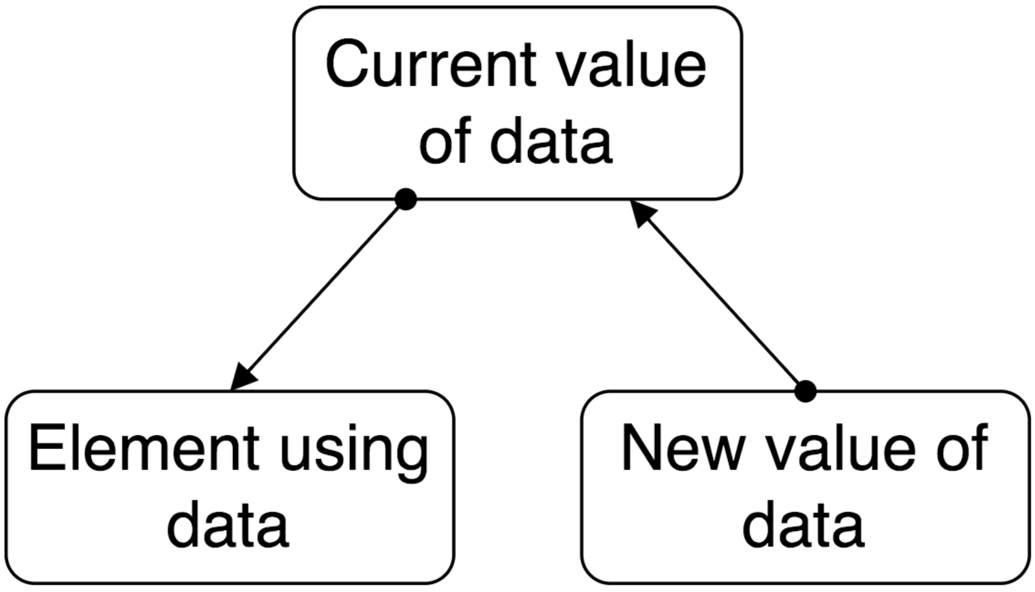


Figure 12: FRP allows us not worry about updating element state directly.

FRP in U·(TP)²

Functional reactive programming is used in our Threepenny-gui implementation of U·(TP)² in two places. We use FRP to manage the current workspace directory and the theory graph. In Figure 13, a single-node theory graph can be seen in our Threepenny-gui implementation of $U\cdot(TP)^2$, the single node is labeled ”\ROOT$0”. The screenshot also shows the current workspace being displayed.

In each of these cases we have a variable, accessible application-wide, which represents the data over time. In the workspace directory’s case the data is a string, in the theory graph’s case it is a tree structure. Whenever a new value of the data is computed somewhere in the application, usually due to user input, we emit that new value. For example when a user selects a workspace, we emit a string, the file path of the workspace directory. If the data’s value has changed, any elements depending on the data will receive the new value and update themselves. This paragraph relates closely to Figure 12.

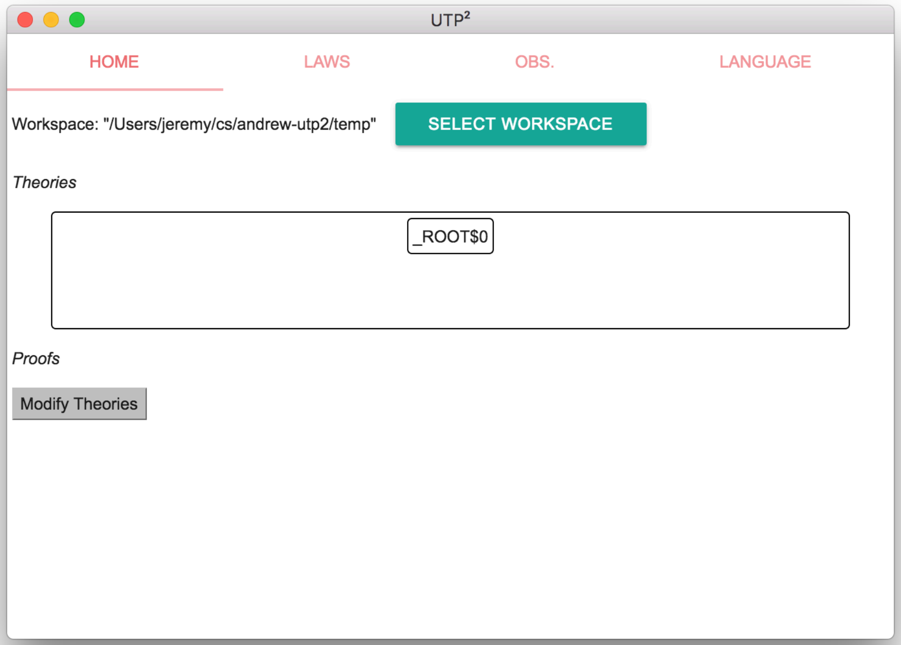

Workspace: "/Users/jeremy/cs/andrew-utp2/temp"

Figure 13: Homescreen of Threepenny-gui implementation of $U\cdot(TP)^2$.

Abstract GUI Layer

Integrating with Existing U·(TP)^2

Reflection

Paragraph on slow development loop where we compile, view changes, edit, re-compile… In contrast with modern web development where we can have hot-reloading of changed modules.

Paragraph on Threepenny-gui being maintainable due to small code base, and solving problems of traditional GUI libraries. Strong future. Mention leveraging webdev libraries.

Paragraph on goal accomplishment.

Appendices

tododile threepenny-gui-contextmenu

threepenny-gui-flexbox

Here follows the README of the Haskell package threepenny-gui-flexbox, compiled from GitHub markdown to LaTeX using `pandoc`.

tododile Gorgeous line separator.

Threepenny-gui Flexbox

circleci passing hackage v0.4.2 stackage nightly 0.4.2 stackage lts-8 0.3.0.2

Flexbox layouts for Threepenny-gui.

This library was written following the wonderful A Complete Guide to Flexbox and using the equally wonderful Clay library as a CSS domain specific language.

Usage

- Properties

  Ultimately we just want to set Flexbox properties on elements, both parent and child elements. In CSS these properties would look like `flex-grow: 1;`.

  We collect Flexbox properties that apply to the parent element, things like `flex-direction`, in a `ParentProps` data type. Flexbox properties that apply to child elements, things like `flex-grow`, are collected in a `ChildProps` data type.

If you want `ChildProps` with `flex-grow: 1;` you can just do:

```
flexGrow 1
```

You can define multiple properties using record syntax:

```
order 1 { cflexGrow = 1, cFlexShrink = 2 }
```

Note that in the examples above we used `flexGrow` and `order` to return `ChildProps` with given values set but also with default values set for all other Flexbox properties, unless record syntax is used to override a property.

Some properties like `flexGrow` simply take an `Int` but others take a value from the `Clay` library. Here's an example for `ParentProps`:

```
display Clay.Display.inlineFlex { pFlexWrap = Clay.Flexbox.nowrap }
```

If you just want `ParentProps` or `ChildProps` with default values:

```
parentProps :: ParentProps
childProps  :: ChildProps
```

- Setting Properties

  Once you have your properties defined you'll want to apply them to elements. For this you can use `setFlex` which can be used with Threepenny's reverse function application operator `#`:

  ```
  UI.div # set UI.text "foo" # setFlex (flexGrow 1)
  ```

  You can also convert `ParentProps` or `ChildProps` to a `[(String, String)]` which is how Threepenny expects CSS. This can be done using `toStyle` which is defined in the typeclass `ToStyle`:

  ```
  UI.div # set UI.style (toStyle $ order 1)
  ```

- 'flex'

  We provide a utility function `flex` (and a few variants thereof) which takes both parent and child elements and their respective `ParentProps` and `ChildProps`, applies the properties to the respective elements and then returns the parent element with children attached.

Here is a full example, which produces the above image of three orange text boxes in ratio 1:2:1. First done without `flex_p` and then with `flex_p`. `flex_p` is a variant of `flex` which applies default Flexbox properties to the parent element.

```haskell
-- |Example without 'flex_p'.
example :: Window -> UI ()
example w = void $
  getBody w # setFlex parentProps #+ [
      foo # setFlex (flexGrow 1)
    , foo # setFlex (flexGrow 2)
    , foo # setFlex (flexGrow 1)
    ]

-- |Example with 'flex_p'.
example' :: Window -> UI ()
example' w = void $
  flex_p (getBody w) [
      (foo, flexGrow 1)
    , (foo, flexGrow 2)
    , (foo, flexGrow 1)
    ]

-- | Simple coloured 'div'.
foo = UI.div # set UI.text "foo"
             # set UI.style [("background-color", "#F89406"),
                             ("margin", "8px")]
```

Pull Requests and Issues

tododile Link and sentence on each.