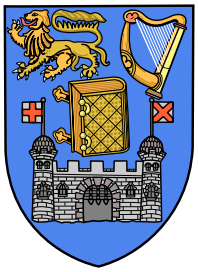


Declaration

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

Table of Contents

Part I

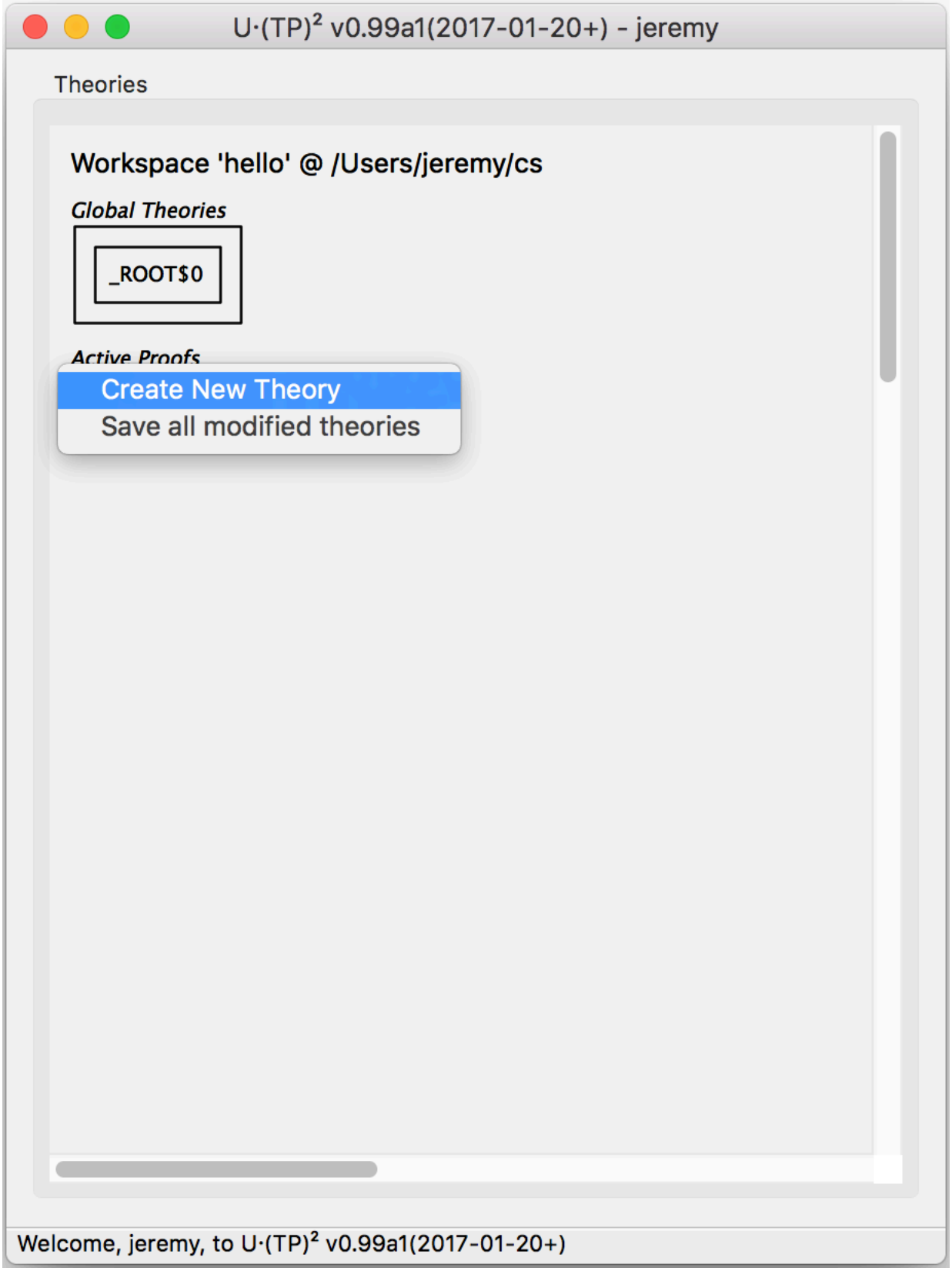
Background

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.

TODO title


U·(TP)² v0.99a1(2017-01-20+) - jeremy
Theories
Workspace 'hello' @ /Users/jeremy/cs
Global Theories
_ROOT$0
Active Proofs
Create New Theory
Save all modified theories
Welcome, jeremy, to U·(TP)² v0.99a1(2017-01-20+)

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

Project Justification

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 wouldn't 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 any Stack resolver. 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 a Stack resolver 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 isn't **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 U·(TP)² could be built as OS native packages e.g. `.deb` packages for Debian or `.app` for macOS, or at least executables that could be distributed. This proved difficult for the existing project as it was not being successfully built on macOS and was difficult to build on Linux.

> 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

Chapter 3

Third chap