Graphical User Interfaces in Haskell

Good morning, my name is Gideon Sireling and I'll be presenting my project on graphical user interfaces in Haskell.

Outline

After introducing the topic with some background material, I'll analyse past work on the topic, and define the project's aims going forwards. I'll then present the design and implementation of my solution, how it was tested, and the project's conclusions. Finally, we can evaluate what has been achieved, and present some suggestions on how it could be extended.

Introduction

Programming languages may be broadly divided into two classes; imperative and functional. The languages typically used in business environments, such as Java and C++, are imperative; a program consists of a series of statements that perform IO, or operate on the data in the computer's store. Functional languages, on the other hand, such as Haskell and ML, are declarative; the program is expressed as a set of immutable equations, and is executed by evaluating the main equation.

Functional languages are typically found in computer science departments and financial firms. However, interest in functional programming outside of these niches is growing. Immutable data structures and referential transparency eliminate many of the difficulties associated with parallel processing, essential for utilising the capabilities of multi-core architectures. Similarly, the strong type systems of Haskell and ML enhance confidence that the code will perform as the programmer intended.

Graphical user interfaces have long been a strength of imperative, object-oriented languages; graphical widgets are easily modelled by stateful objects. If functional languages are to gain traction as mainstream programming languages, they will have to work with tasks outside of their traditional domains.

The aim of this project will be to investigate how to better support GUI development with the purely functional language, Haskell.

Literature Review

"Purely functional" means that all Haskell functions are referentially transparent; a function is guaranteed to always give the same output for a particular set of inputs. This presents a problem for IO; how can we have a function to, for example, read from the keyboard, which will give a different result each time it is called?

The solution to this is monads, a construct borrowed from category theory. A proper treatment of monads is beyond the scope of this overview, but it suffices to say that programming in the IO monad gives a reasonable illusion of imperative code.

So the most obvious solution for GUI programming is to bind to an existing graphical toolkit, and write the GUI code as IO actions. This has been done with the three most popular toolkits; Gtk2Hs for GTK+, WxHaskell for WxWidgets, and QtHaskell for Qt.

An alternative approach is to find a more functional, or declarative abstraction for constructing GUIs. This is the direction taken by functional reactive programming. The fundamental concepts are events, which indicate that something has occurred at a specific time; behaviours, which are time-varying values, and reactive behaviours, which are behaviours that change in response to events. The graphical output consists of widget properties bound to behaviours. So, for example, in a very simple FRP program where a label widget shows what's been typed at the keyboard, key presses are events, the accumulation of everything that's been typed is a reactive value, and the text property of a label widget is bound to this value.

Analysis

So given that functional reactive programming is a much more functional paradigm than the imperative toolkits, we would expect FRP to dominate in graphical Haskell applications. However, the opposite is true. FRP driven user interfaces only seem to exist in the sample code that comes with FRP libraries, whereas real-world programs all use one of the imperative toolkits.

We can understand why this is by considering the domains in which FRP has developed. The most prominent framework, Yampa, grew out of a robotic control system, while the graphical frameworks are mostly focused on computer-driven animations. Along with arcade games, these systems all involve autonomous sensors and processors communicating with each other through a network of event streams. FRP is an excellent model for these kinds of scenarios, but does not deal with global state.

If you think of a typical line-of-business application, where the user enters and edits data, and displays reports, this is a very imperative workflow. It follows that the imperative IO monad is perfectly adequate for modelling this workflow. However, libraries could be written to ease the execution of particular tasks, with IO actions holding everything together.

Aims

For this project, I'm going to develop a framework for data binding, a key enabler of Rapid Application Development. The simple case of one-way data binding has a widget continuously updated to reflect some mutable value, such as whether an application is online. Two-way data binding is ubiquitous in data processing applications, where form fields are updated to reflect the value of a database record, and user edits to these fields are posted back to the database.

The design will be guided by three principles:

- Existing work will be reused wherever possible
- The framework will be simple to use
- and the framework will live in the IO monad

Design

The framework has five key components:

- An interface for mutable variables
- A data type to describe a binding
- Simple data sources, which can be bound to IO objects
- Binding lists, which bind a list of data

- and a common binding interface for all data sources

MVP

The binding framework is based on the Model View Presenter pattern. The model is the binding source's data source, which is presented to the graphical view by a binding. The view is able to send updates back to the model.

Class Diagram

Here you can see how the different parts of the design fit together. The binding source encapsulates a variable, which holds the mutable data; this data can only be accessed through the source's own variable interface. The source also holds a collection of bindings, each of which can apply the data to an IO target.

Sequence Diagram

This sequence diagram shows the process for updating a binding source. The binding source is updated through its variable interface, which then passes on the update to its encapsulated variable. It then instructs each of its bindings to update their targets with the new data.

Packaging

All this functionality is agnostic to the type of the binding target, and forms the binding-core package. Convenient toolkit-specific functions for binding to graphical widgets are provided in the binding-gtk and binding-wx packages.

Testing

Two types of testing were used to ensure correctness; unit testing and integration testing.

Unit testing tests one function at a time, verifying that the actual output for a given input is the same as the expected output. Integration testing verifies that a system's components work together, and with their environment. The binding-core package comes with full unit test coverage. The binding-gtk and binding-wx packages are targeted at integrating the core framework with graphical toolkits, so they're packaged with integration tests.

There are two major unit testing frameworks for Haskell; HUnit and QuickCheck. HUnit is a member of the traditional xUnit family, where each test asserts that a function's actual output matches the expected output for a given input. QuickCheck, on the other hand, offers a more declarative approach; the programmer lists various properties which the function must adhere to, and QuickCheck generates the test data at run time.

For the integration tests, two programs are provided in each package; one to demonstrate simple data binding, and one to demonstrate binding lists.

Conclusions

So what conclusions can we take away from this? We've seen that there's no one-size-fits-all abstraction for GUI work; rather, a different abstraction is needed for each type of GUI task, with the IO monad holding everything together. Functional reactive programming provides an abstraction for animations and arcade games; a data binding library provides the support needed for writing data

processing applications. Finally, we can conclude that Haskell as a language is perfectly capable for general business software; it just needs the appropriate tools and libraries.

Evaluation

The project began with a somewhat quixotic quest for one abstraction to rule all Haskell GUI development. After reviewing the problem space and the current state of solutions, it was found that a more pragmatic approach would be to identify specific domains that could do with better support, such as data binding, and write libraries specific to these domains. Having developed a library for data binding with Haskell, there is every reason to expect that Haskell will prove equally capable with other requirements for general business programming, including those traditionally considered to be the sole domain of imperative languages.

Future Work

Looking forward to how this work could be extended, the next step might be what Microsoft terms “complex data binding”. Instead of a binding list being bound to a set of widgets which show one item at a time, it could be bound to a grid, which shows all the items simultaneously. Moving on, we would like to support relationships between data sources; selecting a customer from one binding list should load all the customer’s orders from a related binding list into a grid.

This project has only tackled the GUI aspect of data binding. The other side of the data binding coin is automatic persistence. .NET, Ruby, and other languages provide facilities whereby a data set can be automatically loaded from a persistent store, edited by the user, and the changes posted back to the store.

Persisting objects, the standard data type of object-oriented languages, in a relational database is the domain of object relational mappers. This has proven extraordinarily difficult to get right in practice, and has been termed “The Vietnam of computer science”. Functional data structures are much more compatible with the relational model; perhaps Haskell could be better suited to data processing applications than object-oriented languages?

Summary

Imperative languages are commonly considered to be the only solution for developing business software, while functional languages are relegated to academia and financial wizardry. However, more general purpose programming is now finding that it needs some of the benefits which functional languages can bring; easy parallelism, and strong static guarantees of correctness. If functional programming is to go more mainstream, it will have to tackle common programming tasks which it has hereunto neglected. We have shown how Haskell, a purely functional language; can adequately bind data to a graphical user interface; and by extension, can be considered a suitable language for general business programming.