Graphical User Interfaces in Haskell

by Gideon Sireling

August 2011

Abstract

Graphical user interfaces (GUIs) are critical to user friendliness, and are well supported by imperative, particularly object-oriented, programming languages.

This report focuses on the development of GUIs with the purely functional language Haskell. We review prior efforts at supporting such interfaces with functional idioms, and investigate why these are rarely used in practice. We argue that there is no overarching solution, but rather, that each class of graphical application should be supported by a domain-specific abstraction.

Finally, we present such an abstraction for data-processing applications; a framework for binding data to graphical interfaces. The framework does not attempt to replace existing graphical toolkits; rather, it adds a new layer of abstraction, with interfaces provided to Gtk2Hs and WxHaskell. Simple examples demonstrate how Haskell can be utilised to accomplish this task as easily as any imperative language.

Acknowledgement

I would like to thank my supervisor, Mr Andy Symons, for the clear direction which he provided throughout this project.

I would also like to thank the good folks who so freely donate their time and expertise to the Haskell Platform, EclipseFP, and all the community projects utilised in this work.

Finally, I wish to express my deepest gratitude to my family, who tolerated all the time invested in this project with the utmost grace and patience.

Table of Contents

1 Introduction

Programming languages may be broadly categorised as imperative or functional (although many languages support elements of both). These are two fundamentally different approaches, and a concept which is easily expressed in one paradigm may be troublesome in the other. This project focuses upon a functional language and how it can be extended to enable graphical user interfaces.

1.1 Imperative Programming

All the commonly used programming languages (C, C++, C#, PHP, Perl etc.) are imperative. Imperative languages are executed as a series of statements, which typically read a value from memory (or input), may process the value in some way, and then write it back to memory (or output). Variables represent memory locations, and are naturally mutable. A function is a lexically scoped sequence of statements, which may be given some values as input, and may return a value to the caller as output.

Imperative languages model the computer's von Neumann architecture. As such, they are an excellent abstraction of the underlying hardware's operation.

1.1.1 Object Orientation

An important abstraction supported by many imperative languages is object orientation. A *class* defines a set of variables and methods. An *object* is a specific instantiation of a class, hence two objects of the same class will have the same variables and methods, but their variables may hold different values. A class may derive from another class. The child will inherit all of the parent's members, and may define some more of its own. (There are other approaches to object orientation, but this very brief explanation should suffice for a general overview.)

1.2 Functional Programming

Imperative programming may be considered the practical approach to programming. It encodes simple CPU operations as statements, then organises them with convenient structures (e.g. loops, procedures and objects). Functional programming, on the other hand, is the theoretical approach. Functional programming languages, such as Haskell and ML, are based on the Lambda calculus, a mathematical theory of computation. A functional program is executed by evaluating a function, which may itself be composed of other functions.

1.2.1 Pure Functions and IO

A *pure function* adheres to the mathematical definition of a function; it is a static mapping of elements from the function's domain to its codomain. This creates a problem for interactive systems. A pure function is referentially transparent, meaning that it will always give the same output for a given input. How, then, can a program change its behaviour in response to an interactive environment?

Haskell's solution is the Monad [**1**], discussed further in 2.2. Monadic programming, along with syntactic sugar, can make pure IO look very much like imperative programming:

```
do putStrLn “What is your name?”
   name <- getLine
   putStrLn $ “Hello ” ++ name
```

1.3 Graphical User Interfaces

An application's graphical user interface (GUI) comprises graphical elements such as windows, buttons and icons, with which the user interacts by means of a pointing device (such as a mouse) and other input devices. A well-designed graphical interface is much more appealing and user-friendly than a character-based terminal interface, but also much harder to design and implement [**2**].

1.3.1 Graphical User Interfaces in Haskell

Object oriented languages excel at GUIs, and this has been a major driver in their adoption. The graphical elements of a GUI, with their mutable state, events, and taxonomy, map naturally to object orientated classes and inheritance.

The most obvious technique for programming GUIs in Haskell is the IO Monad. Unfortunately, while this suffices for simple interactions, graphical user interfaces are particularly challenging. A graphical application may deal with hundreds of graphical objects, each of which has a complex state and many events. Lacking object orientation, such an application will resemble procedural C code, with thousands of lines of imperative IO statements.

1.4 Project Aims

Haskell is primarily an academic language; its use in business tends to be limited to niches such as finance, which benefit from a functional approach and the safety of a strong, expressive type system.

However, broader interest in functional programming is growing. Modern software is exceedingly complex, and the interaction of program components with the global environment is a common source of intermittent bugs, which cannot be reliably reproduced for analysis. Additionally, modern personal computers have multiple processing cores, and an application must execute multiple computations in parallel to utilise them. Pure functions solve both of these problems, as they are insulated from the global environment, and can be evaluated in isolation.

Rich user interfaces are essential for a broad class of personal and business applications. While some work has been done on programming GUIs in Haskell, the standard approach is to use the IO monad, and program in imperative style.

The aim of this project is to critically review past efforts at developing GUI functionality for Haskell, and investigate how such functionality can be better supported. It is hoped that an examination of different approaches and their successes will highlight how the effort can be most effectively brought forward.

2 Overview of Haskell and Graphical User Interfaces

We will begin with a brief overview of Haskell, concentrating on the features that will be used over the course of the project, and the current state of Haskell GUI programming.

2.1 Haskell Language Concepts

In 1987, the conference on Functional Programming Languages and Computer Architecture (FPCA '87) decided to create a new, purely functional language, to consolidate the much-duplicated research that was going on in several existing languages [**3**]. This language became Haskell.

The Haskell language is defined in the Haskell report, the current version of which is Haskell 2010 [**4**].

2.1.1 Values

Haskell values include literals (such as numbers and strings), variables, expressions, and functions. For example,

- Literals: `23`, `“hello”`
- Expression: `x + y`
- Function: `\x -> x + 1`

2.1.2 Variables

A *variable* is an identifier which is bound to a value, e.g.

```
x = 3
```

Variables are immutable; once bound, the value of a variable cannot be changed.

All variables begin with a lower case letter.

2.1.3 Types

Haskell is statically and strongly typed. Static typing implies that the type of every value (e.g. literals, function arguments, expressions) is known at compile time. Strong typing implies that attempting to use the wrong type (e.g. `sqrt "string"`) will be prevented by the compiler.

Types can be given synonyms with the `type` keyword, such as

```
type Point = (Double, Double)
type Transform = Point -> Point
```

2.1.4 Algebraic Data Types

New data types are defined with the `data` keyword, and are constructed with a type constructor. For example, a new Boolean type may be declared with

```
data Bool = True | False
```

This declares a type called `Bool` (the type constructor) with two data constructors; `True` and `False`. The data constructors may then be used wherever a value is expected, e.g.

```
t = True
f = False
```

Data constructors may take other values as arguments, e.g.

```
data Point = Point Double Double
p = Point 4.5 7.8
```

Type and data constructors are capitalised.

2.1.5 Polymorphic Types

A type is polymorphic if its data constructor has a parameter which is not of a specific type. For example, the following code declares a type `Pair`, with two type variables, `a` and `b`.

```
data Pair a b = Pair a b
x = Pair 3 "string" :: Pair Int String
y = Pair 4.5 'c' :: Pair Double Char
```

2.1.6 Type Annotations

Also shown in this example are type annotations (the lines containing ::). The compiler can usually infer the correct type, in which case type annotations are optional. However, they may aid comprehension by explicitly indicating an expression's type.

The compiler will always ensure that type annotations are correct, which can catch errors where an expression does not mean what the programmer intended it to mean.

2.1.7 Functions

A *function* is an expression which maps one or more parameters to a value. For example,

```
\a b -> a + b
```

is an expression which returns the sum of two numbers. The expression may be bound to a variable for reuse, e.g.

```
add = \a b -> a + b
```

A more intuitive synonym for this binding is

```
add a b = a + b
```

Functions are pure, i.e. referentially transparent. Evaluating a function over a particular set of values will always yield the same result. This is different to functions in an impure language, which could ignore their input and return a random number.

Sum may be given a type annotation such as

```
add :: Int -> Int -> Int
```

This defines sum as taking two integers, and returns an integer.

2.1.8 Currying and Partial Application

A simple function to add two integers may be defined as

```
add :: (Int,Int) -> Int
add (a,b) = a + b
```

This is a straightforward function that takes a pair of integers, and returns their sum. However, there is an advantage to be gained with the more common style of definition:

```
add :: Int -> Int -> Int
add a b = a + b
```

This style of function definition is called currying. It defines a function which takes *one* argument, and returns a function that takes a further argument; it is this second function which calculates the sum. To make this more explicit, note that type signatures are right-associative, hence the type of add can also be written as

```
add :: Int -> (Int -> Int)
```

Applying this function to two arguments is evaluated thus:

```
add 1 2 =
(\a -> \b -> a + b) 1 2 =
(\b -> 1 + b) 2 =
1 + 2 =
3
```

It is perfectly acceptable to apply add to just one argument. For example, we can create a function which adds 3 to its argument with

```
add3 :: Int -> Int
add3 = add 3
```

2.1.9 Type Classes

It is often desirable for functions to be overloaded across many different types. For example, the equality operator works on strings, numbers, and many other types. This is achieved by declaring a *type class* such as

```
class Eq a where
    (==) :: a -> a -> Bool
```

This declares a type class Eq. A type a is an instance of Eq if there is an == operator that takes two values of the type, and returns a Bool. (The parenthesis around == declare it to be infix.)

The class Pair defined above can then be made an instance of Eq with

```
instance (Eq a, Eq b) => Eq (Pair a b) where
    (Pair a b) == (Pair c d) = (a == c) && (b == d)
```

We first state that if a and b are instances of Eq, then Pair a b is also an instance of Eq. We then give the definition of == for a Pair.

2.1.10 Existential Types

Haskell lists are homogenous; all the elements must be of the same type. This means that we cannot write an expression such as

```
map show ["123", 123]
```

even though show "123" and show 123 are both valid individually.

The first stage to solving this is to wrap each element in a custom type, such as

```
data Showable x = Show x => Showable x
instance Show Showable where
```

```
        show (Showable x) = show x
```

We may then attempt

```
    map show [Showable "123", Showable 123]
```

However, this still fails to compile. `Showable "123"` is of type `Showable String`, whereas `Showable 123` is a `Showable Int`; the list is still heterogeneous.

A solution to this is existential types [**5**]. Using the `forall` keyword, we can eliminate the type variable from `Showable`.

```
    data Showable = forall x. Show x => Showable x
```

This tells the compiler that it should neither know nor care what the precise type of the `Showable`'s data is; we only know that is of the class `Show`. The "mixed" list is then valid, and has the type `[Showable]`.

2.1.11 Functors

One of the simpler type classes in the standard library is `Functor`, defined as

```
    class Functor f where
        fmap :: (a -> b) -> f a -> f b
```

A `Functor` is a "container" for values; `fmap` applies an ordinary function to the contents of the container. For example, lists are `Functor`s, where `fmap` is synonymous with `map`.

```
    fmap (+3) [4,5,6] == [7,8,9]
```

2.1.12 Applicative Functors

Functors only contain values, while `fmap` applies an ordinary function to them. Applicative Functors [**6**] extend this by storing the function in the `Applicative` as well.

```
    class Functor f => Applicative f where
        pure  :: a -> f a
        (<*>) :: f (a -> b) -> f a -> f b
        (*>)  :: f a -> f b -> f b
        (<*)  :: f a -> f b -> f a
```

pure is used to put a value (or function) in the Applicative Functor, while <*> has a similar role to `fmap`.

```
    pure (+3) <*> [4,5,6] == [7,8,9]
```

> and < are combinators, which ignore their first and second arguments respectively.

2.1.13 Arrows

The `Arrow` family of type classes define computations which can be chained. The current implementation has a large number of classes and functions, but for simplicity's sake, we will work with the original proposal [**7**].

```
    class Arrow a where
        arr   :: (b -> c) -> a b c
        (>>>) :: a b c -> a c d -> a b d
        first :: a b c -> a (b,d) (c,d)
```

arr takes a function which maps b to c, and yields an *arrow*. >>> composes two arrows, in the same way that ∘ composes functions. first transforms a simple arrow into an arrow which accepts a tuple; the first component is processed by the original arrow, while the second component is passed through unaltered.

Arrows have an obvious application to dataflow programming, which will be presented in more detail in 2.6 (Functional Reactive Programming).

2.1.14 Arrow Syntax

GHC provides syntactic sugar for arrows, as originally proposed by Ross Paterson [**8**]. For example,

```
    arrow = proc x -> arr show -< x+1
```

creates an arrow equivalent to

```
    arrow = arr (+1) >>> arr show
```

which will increment its argument, and convert the result to a string. The proc syntax can be used to compose multiple arrows in a do block; this will become useful in Functional Reactive Programming (see 2.6).

2.1.15 Monads

Monads are a subset of arrows, in that they define chainable actions. Hence, Monads can be viewed as a computational strategy, defining how a set of computations is to be combined.

The Monad class is defined thus:

```
    class Monad m where
        (>>=)  :: m a -> (a -> m b) -> m b
        (>>)   :: m a -> m b -> m b
        return :: a -> m a
        fail   :: String -> m a
```

Inspired by the monads of category theory, a monad defines a container (m) for computations. a -> m b is the type of a monadic function (action) which takes a value a, and returns a value b, inside the monad m. The bind function >>= combines two actions, extracting the value returned by the first action, and feeding is a parameter to the second action. >> is a bind which ignores the output of the first action, and chains a second action which does not take input. return injects a value into the monad. fail is a convenience function which is not actually related to the definition of a monad; it defines an error handling mechanism for failed actions.

For example, the Maybe monad is defined as

```
    data Maybe a = Nothing | Just a
    instance Monad Maybe where
        (Just x) >>= k = k x
        Nothing  >>= _ = Nothing
        (Just _) >>  k = k
        Nothing  >>  _ = Nothing
        return = Just
        fail _ = Nothing
```

`Maybe` is a wrapper for computations which may not return a result. If a `Maybe` action succeeds, it returns `Just value`; otherwise it returns `Nothing`. The bind function will feed the contents of `Just` to the second action, and short-circuit to `Nothing` if the first action failed. `return` wraps a value in `Just`, while failure is indicated with `Nothing`.

Thus, the Maybe monad enables a sequence of functions to bail out with Nothing if any function in the sequence returns Nothing, without ever having to explicitly check for Nothing. This is a great improvement on the usual situation in imperative programming, where the return value of every statement must be tested for null and explicitly handled, and where failure to do so is a frequent cause of program crashes.

2.1.16 do Notation

A succession of arrow compositions or monadic computations may be listed in a `do` block. For example, the marketing department targets campaigns by postcode. We want to find a particular customer's postcode, the campaign associated with this postcode, the marketing manager in charge of this campaign, and finally, the manager's mobile number. Any of these lookups could fail (e.g. not all campaigns are assigned to managers, and not all managers have company phones.) The following code will return `Just mobile` if the number can be found. If any of the lookups fail, it will return `Nothing`.

```
do
postcode <- lookup name customers
campaign <- lookup postcode campaignRegions
manager <- lookup campaign campaignManagers
lookup manager mobiles
```

This is equivalent to

```
lookup name customers >>= \postcode ->
lookup postcode campaignRegions >>= \campaign ->
lookup campaign campaignManagers >>= \manager ->
lookup manager mobiles
```

2.1.17 Modules

If a code file is intended to be a reusable module, it begins with the `module` keyword, followed by the module's name. Modules are imported with an `import` directive.

The standard library defines a module called `Prelude`, which is implicitly imported into every code file.

2.2 The IO Monad

The original motivation for monads was IO. Reverential transparency requires the result of a function to depend only on its parameters, not the state of disk files or other input mechanisms in the program's environment. Another problem is that Haskell's laziness makes it difficult to control the order in which functions are evaluated; the program may attempt to read from a file before it has been written, or wait for the user's input before asking them for it.

The solution adopted by Haskell is that an IO function conceptually takes the state of the world as an input, and returns the new state of the world in its output. Thus, referential transparency is preserved, and since the state returned by an IO action will be used for the input of the next action, correct ordering is guaranteed. The type of an IO action is defined by GHC as:

```
        State RealWorld -> (State RealWorld, a)
```

The actual implementation of `State RealWorld` and the bind functions are handled in the compiler.

2.2.1 IORef

Some requirements are considerably inconvenient to satisfy with immutable variables. For example, callbacks in an event-driven UI may need to read and write to the global application state. In such cases, an `IORef` can be used. An `IORef` represents a pointer to a pure value. `newIORef`, `readIORef`, and `writeIORef` are straightforward functions for creating an `IORef` wrapper, reading the contents of the wrapper, and writing to it. All these functions are IO actions; writing to and reading from memory is semantically no different from disk IO.

2.3 Implementations of Haskell

Many Haskell compilers have been written, each with their own language extensions and libraries. Hugs [**9**], for example, is popular for teaching and casual use, while the York Haskell Compiler [**10**] and Utrecht Haskell Compiler [**11**] are more research orientated.

2.3.1 The Glasgow Haskell Compiler

The most popular Haskell compiler for software development is the Glasgow Haskell Compiler (GHC) [**12**], which provides many useful extensions, and emits aggressively optimised binaries for a variety of platforms.

Many modern Haskell libraries can only be compiled with GHC, including the GUI Toolkits soon to be surveyed (Gtk2Hs, WxHaskell, and QtHaskell). Thus, GHC is the only appropriate implementation for this project. Furthermore, none of the features or extensions supported by other compilers are relevant to graphics work.

2.3.2 Packages

Cabal is a suite of Haskell tools to facilitate code reuse. Modules are zipped with a descriptive metafile and configuration information to form a *package*. Cabal can fetch packages from an online repository and install them, along with all their dependencies.

2.3.3 Hackage and the Haskell Platform

Hackage [**13**] is a public package repository. There are many libraries available for a wide variety of tasks, along with documentation (generated by Haddock [**14**]) and build reports.

The Haskell Platform [**15**] is a convenient installation package for GHC, Cabal, the most important libraries from Hackage, and a suit of development tools. The remainder of this report will use the Haskell Platform.

2.4 Noughts and Crosses Example

Noughts and crosses is a simple pencil and paper game played on a 3 x 3 grid of squares. The first player draws an X in any square. The second player then draws an O in any empty square. The two players continue to take turns until one of them has a line of three tokens, thereby winning the game. If the board is filled without a win, the game is a tie.

This simple, interactive game will be used to demonstrate the various GUI toolkits and frameworks (barring those which are no longer maintained). All the source code is given in Appendix A.

- Module `OX` contains the game's data types and logic.
- Each GUI is contained in its own module, which imports `OX`.
- Module `Console` is a textual version of the game for comparison. [1]

In order to exercise the frameworks a little harder, the board will be made up of three kinds of widgets. The first row is buttons, the second row is radio buttons, and the third row is made up of check boxes. (The resulting interface is exceedingly unappealing, but aesthetics are orthogonal to our functional aims.)

The graphical examples will be expanded on throughout the remainder of this chapter. Figure 1 demonstrates a game of noughts and crosses in the console.

```
  1 2 3
 +-+-+-+
1| | | |
 +-+-+-+
2| | | |
 +-+-+-+
3| | | |
 +-+-+-+

Turn: X
```

```
(1,1)

  1 2 3
 +-+-+-+
1|X| | |
 +-+-+-+
2| | | |
 +-+-+-+
3| | | |
 +-+-+-+

Turn: O
```

```
(2,2)

  1 2 3
 +-+-+-+
1|X| | |
 +-+-+-+
2| |O| |
 +-+-+-+
3| | | |
 +-+-+-+

Turn: X
```

```
(3,1)

  1 2 3
 +-+-+-+
1|X| |X|
 +-+-+-+
2| |O| |
 +-+-+-+
3| | | |
 +-+-+-+

Turn: O
```

```
(2,3)

  1 2 3
 +-+-+-+
1|X| |X|
 +-+-+-+
2| |O| |
 +-+-+-+
3| |O| |
 +-+-+-+

Turn: X
```

```
(2,1)

  1 2 3
 +-+-+-+
1|X|X|X|
 +-+-+-+
2| |O| |
 +-+-+-+
3| |O| |
 +-+-+-+

X won!
```

Figure 1: Noughts and Crosses for Teletypes

[1] The code to render the game state as a string is included in OX, as it depends on the internals of encapsulated data types.

2.5 GUI Toolkits

The existing solutions for developing graphical user interfaces in Haskell fall into two categories, which may be (somewhat arbitrarily) termed toolkits and frameworks. Toolkits are lower level libraries, which enable imperative-style user interface programming in Haskell. Frameworks are much more ambitious; they use a high-level abstraction of user interfaces to enable a more declarative, functional style of programming.

There are a number of cross-platform graphical toolkits, written in C or C++. These toolkits provide an Application Programming Interface for creating and managing windows, buttons, and other graphical widgets. They often provide additional functionality, such as networking, but this is not relevant to our investigation.

The more common toolkits have bindings enabling them to be used from numerous languages. These bindings provide wrappers in the host language that enable the toolkit’s C/++ functions to be utilised without the programmer having to marshal data across language boundaries, or concern themselves with C/++ idiosyncrasies.

In the case of another imperative language, such as Java or Ruby, this mapping is fairly straightforward; C functions (and C++ objects) map to functions and objects in the host language, while the wrappers mostly deal with marshalling data types. Haskell, however, as a purely functional language, presents problems in this regard. The language does not support object-orientation, variables are immutable, and IO functions are a special facility which must be “caged” in the IO monad.

The Haskell toolkit wrappers simply translate the underlying API methods into IO actions, using custom data structures to manage widget handles.

2.5.1 Gtk2Hs

One of the most popular cross-platform graphical toolkits is GTK+; its Haskell binding, likewise the most popular toolkit for Haskell applications, is Gtk2Hs.

2.5.1.1 *GTK+*

GTK+ [**16**], originally the GIMP toolkit, is a cross-platform toolkit for creating GUIs. Originally written for GIMP (the GNU Image Manipulation Program), GTK+ is a set of C libraries for working with graphical widgets and bitmapped text. Implementations are available for many common platforms, and bindings exist to numerous programming languages.

2.5.1.2 *Layout*

Layout is implemented with layout widgets. For example, a `VBox` stacks its children vertically, and a `Table` arranges its children in a grid. Positioning and sizing is automatic; properties on the child widget can request special size or positioning requirements.

2.5.1.3 *Glade*

Glade [**17**] is a user interface designer for GTK+. The programmer arranges widgets with a point-and-click interface, and sets properties governing their behaviour and appearance. Glade saves the UI as an XML file, which GTK+ reads at runtime.

Glade was not used for this project, as it does not build on Windows. This is not a great loss, as the structured, repetitive nature of the board lends itself to easy construction with Haskell's combinators.

2.5.1.4 *Gtk2Hs*

Gtk2Hs [**18**] provides access to the Gtk+ API through idiomatic Haskell functions, which wrap the Foreign Function Interface. A number of utility functions are also provided.

The object-orientated widget hierarchy is modelled with Haskell type classes. Each widget is defined as a newtype (a data type created from an existing type), and a type class. This type class is then declared to be an instance of all its ancestors, and functions are provided to upcast. For example, the hierarchy `Object` -> `Widget` -> `Container` requires the following code:

```
--Object
newtype Object = Object (ForeignPtr Object)

class GObjectClass o => ObjectClass o

instance ObjectClass Object

toObject :: ObjectClass o => o -> Object
toObject = unsafeCastGObject . toGObject

--Widget
newtype Widget = Widget (ForeignPtr Widget)

class ObjectClass o => WidgetClass o

instance WidgetClass Widget
instance ObjectClass Widget

toWidget :: WidgetClass o => o -> Widget
toWidget = unsafeCastGObject . toGObject

--Container
newtype Container = Container (ForeignPtr Container)

class WidgetClass o => ContainerClass o

instance ContainerClass Container
instance WidgetClass Container
instance ObjectClass Container

toContainer :: ContainerClass o => o -> Container
toContainer = unsafeCastGObject . toGObject
```

The complete widget hierarchy requires a vast number of classes and instances, which are automatically generated from the Gtk+ header files.

2.5.1.5 *Attributes*

Widgets are styled and positioned with attributes, such as colour, height, and text.

```
data ReadWriteAttr o a b
type Attr o a = ReadWriteAttr o a a
type ReadAttr o a = ReadWriteAttr o a ()
```

```
    type WriteAttr o b = ReadWriteAttr o () b
```

The basic attribute data type is `ReadWriteAttr o a b`, where `o` is the widget type, `a` is the attribute's `get` type, and `b` is the `set` type. Most attributes are `get` and `set` with the same type, for which the `Attr` type synonym is defined. `ReadAttr` and `WriteAttr` define read- and write-only attributes respectively.

This simple system ensures that all attributes are type-safe, both in their value, and in which widgets they can be applied to. Specifying the value for an attribute (or a transformation to be applied to its existing value) creates an attribute operation (`AttrOp`). Finally, a list of attribute operations is applied to a widget with `set`.

2.5.1.6 *Events and Signals*

Gtk exposes events emitted by the UI subsystem, such as key presses and mouse movements, as *events*. A widget may respond to an event (or a specific set of events) by emitting a *signal*, e.g. to indicate that a button has been pressed. The programmer can respond to events by attaching a callback function as the event/signal handler.

2.5.1.7 *The Game*

Gtk2HS.hs (Appendix A, 7.3) imports the game logic from `OX`, and creates a GTK+ user interface, wiring up Gtk2Hs events to pure functions from `OX`. The state of the game is kept in a mutable `IORef`.

First, the `IORef` `game` is initialised with a new game. Then the widgets are all created, including the layout widgets `vbox` and `table`. `event` is defined to handle widget events; this reads the current state of the `game`, labels the square with the appropriate token, and runs the move through `OX.move`. If the game has been won, an appropriate dialogue is displayed; otherwise, the game state is written back to the `game`, the UI is updated, and play continues.

Finally, `event` is attached as an event handler, and the widgets are placed in their layout containers.

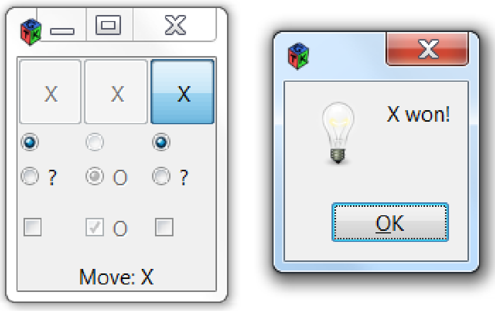

Figure 2: Gtk2Hs

2.5.2 WxHaskell

Another popular cross-platform toolkit is WxWidgets, wrapped by the WxHaskell library.

2.5.2.1 *WxWidgets*

WxWidgets [**19**] [**20**] is a cross-platform toolkit written in C++, which aims to give a native look and feel on each platform. Whereas other toolkits paint their own widgets, wxWidgets uses the platform's native widgets wherever possible. Applications written with wxWidgets thus behave consistently with whichever platform they are deployed on.

2.5.2.2 *Layout*

WxWidgets implements layout with the `Layout` data type. The `widget` function wraps a control in a `Layout`; other functions create simple layout objects, such as empty space. Layouts can be enhanced with transformer functions (`Layout -> Layout`), such as `floatCenter` and `expand`. A layout can contain other layouts, e.g. the `grid` function, which creates a layout that arranges an array of child layouts in a grid. To display a layout, it is assigned to the `layout` property of the containing widget (e.g. the top-level window).

This is a more complex system than the use of layout widgets, but does result in more concise code.

2.5.2.3 *WxHaskell*

WxHaskell models the WxWidgets class hierarchy with phantom types; this is best explained with an example [**21**]. The data type declarations for `Window`, `Control` and `Button` are:

```
data CWindow a
data CControl a
data CButton a
```

No constructors are given for these data types, thus no values can be created, hence the name. The types are then declared as:

```
type Object a = Ptr a
type Window a = Object (CWindow a)
type Control a = Window (CControl a)
type Button a = Control (CButton a)
```

The inheritance hierarchy is thus modelled with nested type synonyms. This is clearer when the type synonyms are expanded:

```
type Window a = Object (CWindow a)

type Control a = Window (CControl a)
               = Object (CWindow (CControl a))

type Button a = Control (CButton a)
              = Window (CControl (CButton a))
              = Object (CWindow (CControl (CButton a)))
```

Button is hence an instance of `Window`, because

```
Object (CWindow (CControl (CButton a))) = Object (CWindow a)
    where a = CControl (CButton a)
```

This approach has a considerable advantage over GTK+'s, in that it avoids an explosion of empty type classes and conversion functions. However, it cannot model multiple inheritance, which is a key feature of WxWidgets' C++ implementation. Instead, common interfaces are implemented with type classes. For example, every widget with a `text` attribute is an instance of the `Textual` class. An advantage of this is that a function or attribute can be implemented differently for different instances of a class; GTK+ is tied more closely to the object-oriented model [**22**].

WxHaskell is currently based on the older 2.8 version of WxWidgets. Although 2.9 is officially a development release, WxWidgets recommends its use, due to the great number of improvements and enhanced stability.

2.5.2.4 *Variables*

WxHaskell has its own `Var` type for mutable variables. This is a wrapper for Software Transactional Memory [**23**], a technique for handling shared state between concurrent threads. A discussion of concurrency is beyond the scope of this project; otherwise, they can be used in the same manner as `IORef`, which was indeed their implementation in earlier versions.

2.5.2.5 *Attributes*

Attributes are similar to Gtk2Hs attributes, but simpler. `Attr w a` is an attribute for widgets of type `w` and values of type `a`. Attributes are combined with values or transformer functions to create attribute operations in the same manner as for Gtk2Hs, and a list of these operations are then `set`. (Gtk2Hs acknowledges WxHaskell as a source for their attribute implementation in source comments.)

2.5.2.6 *Events*

WxHaskell's on function converts events into attributes, which are then `set` in a list like any other attributes. This makes for very concise code, where attributes and event handlers are all set in the widget creation function. Most widgets are instances of the `Selecting` or `Commanding` type classes, which declare the `select` and `command` events respectively.

2.5.2.7 *The Game*

WxHaskell.hs (Appendix A, 7.4) is very similar in structure to Gtk2HS.hs, the major difference being the absence of layout widgets. In their place, all the positioning is done by the `layout` property of the main window. For a more detailed comparison of WxHaskell and Gtk2Hs, see [**22**].

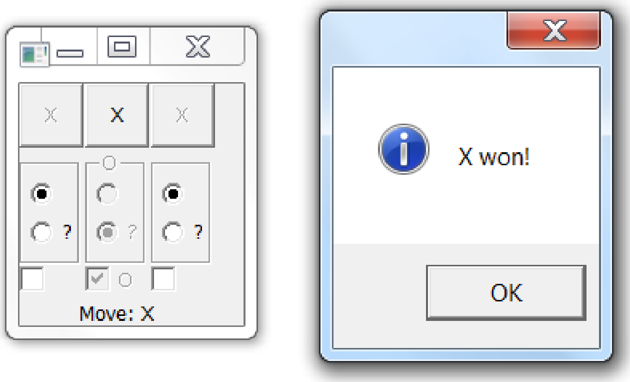

Figure 3: WxHaskell

2.5.2.8 *WxGeneric*

WxGeneric (formerly AutoForms [**24**]) is an interesting extension to WxHaskell. It defines a function `genericWidget`, which given a value, will construct a widget for editing it. This is not of much use in the noughts and crosses example, as a player cannot arbitrarily edit the board. Neither is it simple to create an arbitrary board editor, as WxGeneric would have to be taught how to display the board array as a fixed grid of widgets. Instead, the WxGeneric example (Appendix A, 7.6) generates a form for editing a hypothetical student record.

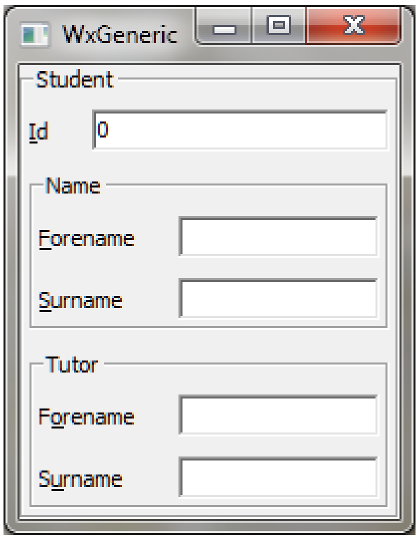

Figure 4: WxGeneric

2.5.3 QtHaskell

Finally, we will examine the third popular choice for cross-platform framework GUI programming, Nokia's Qt.

2.5.3.1 *Qt*

The Qt cross-platform framework [**25**] [**26**] [**27**] is primarily directed at GUI developers, but also contains libraries for networking, database querying, and many other functions. Qt is written in C++, but requires several extensions to the standard language. Qt source files must be pre-processed with the proprietary meta-object compiler before they can be compiled with standard C++ tools. Other proprietary features are implemented with standard C++ macros.

2.5.3.2 *Signals and Slots*

In addition to a regular event system, Qt implements the Observer pattern with *signals* and *slots*. A `QObject` descendant may declare a block of methods to be signals, and another block to be slots. The `connect` method is then used to connect a signal to a slot. When the signal is activated, the slot will be called. Many signals may be connected to the same slot, and many slots may be connected to a single signal. Signals may also be connected to other signals.

2.5.3.3 *Layout*

Widgets are laid out by decedents of the `QLayout` class. Basic layout objects such as `QHBoxLayout` and `QGridLayout` may be nested to create more complex layouts. A layout is assigned to a container widget with `setLayout`.

2.5.3.4 *QtHaskell*

QtHaskell [**28**] provides a somewhat thin Haskell layer over the Qt library. Qt classes become Haskell types, while their constructors and methods become functions. The inheritance hierarchy is modelled by phantom types in the same manner as WxHaskell, and inherited functions become members of type classes. The destruction of Qt objects is in most cases performed automatically when the Haskell value is garbage collected. All QtHaskell functions take a single tuple as the final argument, which contains the arguments to pass to the underlying C++ method.

2.5.3.5 *Slots and Signals: C++ in Haskell*

Signals and slots have no straightforward equivalent in Haskell. If a custom signal or slot is required, the class must first be sub-classed.

1. An abstract data type is declared for the new class.
2. A phantom type synonym is declared, inheriting from the relevant Qt class. QtHaskell provides special sub-classing types for this purpose.
3. A function is written to create a value of the new class, utilising the `qSubClass` function.

The `connectStlot` function is used for connecting a signal to a slot. It requires the following parameters:

1. The object which will emit the signal.
2. The signal's C++ signature, passed as a string.
3. The object containing the slot.
4. The slot's C++ signature, passed as a string.
5. A Haskell function for the slot, which is given the signalling object as a parameter.

An interesting alternative was proposed by Wolfgang Jeltsch [**29**], which converts slots and signals into typed Haskell values. Unfortunately, this approach is only used by HQK library [**30**], which is no longer maintained.

2.5.3.6 *The Game*

The structure of QtHaskell.hs (Appendix A, 7.5) differs from the previous two examples, due to the requirement to subclass the widgets. First, an empty data type is declared for the new widget, and a type synonym given. Then a function is declared to return an instance of the new type. For example, in the case of the button, this is done by

```
data COxQPushButton
type OxPushButton = QPushButtonSc COxQPushButton

oxPushButton :: IO OxPushButton
oxPushButton = qSubClass $ qPushButton " "
```

A further function is then defined to create an instance of the subclassed widget, perform any necessary styling, and attach an event handler as a slot.

The code which calls these functions is generic; each widget emits a `clicked` signal when activated, and the same event handler is attached to each signal. Unfortunately, simply mapping over a list of all the widgets is not straightforward. Haskell lists must be homogenous, but the widgets are of three different types.

To solve this, each such function has the same type, `WidgetCreator`.

```
type WidgetCreator = (forall a.   a -> IO ()) -> IO (QWidget ())
```

Existential Types (2.1.10) are used to hide the type of the button's callback as `forall a,` and the created widget is upcast to a `QWidget`.

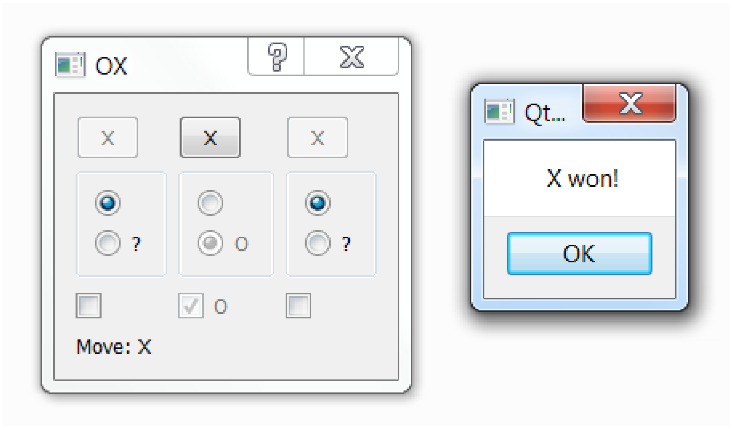

Figure 5: QtHaskell

2.5.3.7 *Imports*

When a Haskell module is imported, GHC links the entire module into the final binary, without eliminating unused code. QtHaskell is an extremely large library, and importing the top level Qt module requires an excessive amount of time for linking. For this reason, only those sub-modules actually used by the code have been imported, instead of the top-level `Qtc` module. (The final binaries are also bloated with much dead code, but this can be `stripped` out.)

2.6 Functional Reactive Programming

Reactive programming is a dataflow programming paradigm. For example, given the expression `a = b + c`, the value of a would continually update to reflect changes in the values of b and c, much like cells in a spreadsheet. *Functional Reactive Programming* simply refers to reactive programming in a functional programming language.

2.6.1 Fudgets

The Fudget Library [**31**], published in 1995, is one of the earliest examples of applying Functional Reactive Programming techniques to implementing GUIs in Haskell. The library is based on *stream processors*, which are arbitrary transformations of an input stream to an output stream. A *fudget* is a stream processor with two pairs of streams; the regular input and output streams which communicate with other stream processors, and the system's IO streams. Stream processors are composed with functions analogous to the Arrows library, which was only proposed five years later.

Fudgets draws its own widgets in X Windows. An interesting feature of Fudgets is that the UI can be laid out automatically from the dataflow graph.

2.6.2 Fran

Fran [**32**] [**33**] is a library for creating interactive multimedia applications. Important concepts introduced by Fran are *behaviours*, *events*, *event streams*, and *reactive behaviours*. Each of these concepts is further explained below.

2.6.2.1 *Behaviour*

A *behaviour* is a value which changes over time, e.g. the coordinates of a bouncing ball.

Behaviours are defined by Fran as

```
type Time = Real
type Behaviour a = [Time] -> [a]
```

Thus, a very simple behaviour, such as a value which increases at twice the rate of time, could be defined as

```
map (*2)
```

2.6.2.2 *Event*

An *event* is a behaviour which occurs at particular times. A trivial example of an event is

```
\t -> if (t > 3) then Just (t*2) else Nothing
```

This is similar to the trivial behaviour we defined above, but only occurs when `time` is greater than 3.

2.6.2.3 *Event Stream*

A sequence of events, e.g. keys pressed, forms an *event stream*. It is defined as

```
type Event a = [Time] -> [Maybe a]
```

An event stream for our trivial event could be defined as

```
map $ \t -> if (t > 3) then Just (t*2) else Nothing
```

2.6.2.4 *Reactive Behaviour*

An event may transform one behaviour into another. This combination of behaviours and events is a *reactive behaviour*. To borrow an example from [**33**],

```
color :: Behavioiur Color
color = red `until` (lbp -=> blue)

circ :: Behaviour Region
circ = translate (cos time, sin time) (circle 1)

ball :: Behaviour Picture
ball = paint color circ
```

First we create a `color` behaviour, which initially has the value `red`. `lbp` is a built-in event indicating that the left mouse button has been pressed; the `until` and `-=>` combinators create a behaviour which is initially red, then becomes blue when the left mouse button is pressed.

`circ` is another behaviour, representing a `Region` in a Cartesian plane. It creates a unit circle, whose position oscillates with time, by means of the built-in `translate` behaviour.

Finally, we create a `Behaviour Picture`, a reactive graphic. The `paint` combinator draws an oscillating red circle, whose colour changes to blue when the left mouse button is pressed.

2.6.3 Yampa

Continuously varying behaviours are semantically sound, but cannot be properly implemented on digital computers. Instead, the behaviour is regularly polled, and the value updated. Clearly, it is desirable to poll as frequently as possible, to reduce the latency between the value of a behaviour changing, and the propagation of the new value to listeners. However, frequent polling consumes computing resources, much of which will be wasted on behaviours whose values have not changed since the last poll. Furthermore, the value of a behaviour may depend on a function over past events. Recalculating this value across the full history of the event stream on every polling interval incurs a considerable time and space cost.

Yampa [**34**] circumvents this by disallowing arbitrary behaviours and events. Instead, the library is based on *signals* and *signal functions*.

2.6.3.1 *Signals*

A `Signal` is a function from time to a value. Signals take the place of behaviours and events in Fran.

2.6.3.2 *Signal Functions*

A signal function (`SF`) is a function which transforms one signal into another, synonymous with Fudgets' stream processors.

`Signal` and `SF` are abstract types; the programmer can only compose existing signal transformers. The library's signal processor knows how to process the basic signals and transformers optimally, avoiding the space and time leaks to which Fran is prone.

Signal transformers are instances of `Arrow`, and are composed with Haskell's arrow syntax. *Arrowised Functional Reactive Programming* (AFRP) is discussed at length in [**35**].

Yampa was designed with a focus on controlling sensors and motors in a robot. However, it is not limited to the domain of robotics; [**36**] demonstrates an arcade game written with Yampa's reactive arrows.

2.6.4 Fruit

Fruit [**37**] [**38**] is a GUI library built on Yampa. Working with Fruit is conceptually similar to Fudgets, substituting signals and signal transformers for streams and stream processors respectively.

A GUI element is defined by Fruit as the signal function

```
type GUI a b = SF (GUIInput, a) (Picture, b)
```

`GUIInput` and `Picture` are input and output IO signals, while the *auxiliary signals* a and `b` are for communication with other signal transformers, enabling composition in the same manner as Fudgets. For example, a button will take mouse or keyboard clicks from `GUIInput`, and change its appearance in `Picture` accordingly. Additionally, the program may instruct the button on auxiliary signal a to enable or disable itself, while the button will inform the program through auxiliary signal `b` when it has been clicked.

A significant advantage of this abstraction is that there is extensive scope for applying higher-level functions to user interfaces. For example, [**37**] demonstrates how multiple views can be easily attached to the same application with Fruit combinators.

Fruit uses Java2D for rendering widgets; the library distribution includes a couple of examples, but these must generally be created by the programmer.

2.6.5 WxFruit

Creating a new graphical widget for Fruit is an extremely laborious process; the widget must know how to process incoming IO events (such as mouse down or key up), and maintain a graphical representation of itself in Java2D. This is not what using a high-level framework is supposed to be about, and prevents the reuse of work in existing GUI toolkits.

WxFruit [**39**] solves this problem by reengineering Fruit to use WxHaskell as a widget library, changing the type of GUI elements to

```
type Widget a b = SF (Event WidgetResp, a) (Event WidgetReq, b)
```

`Event WidgetResp` bears events from the WxHaskell widget, such as 'click' for a button or 'select' for a check-box. `Event WidgetReq` requests the widget to change its state, e.g. new text for a label. WxFruit instructs Yampa to resample the behaviours as soon as a WxHaskell event is received, eliminating latency.

The WxFruit library was published as a proof of concept, implementing a small selection of widgets, and very few events. The problems encountered in extending these are discussed in section 2.6.6 regarding WxFroot.

2.6.5.1 *User Defined Widgets*

What if the user wants to define a widget not available in WxHaskell? In this case, the raw IO events and drawing primitives from which WxFruit shields the programmer are actually needed. For this reason, the element type must become

```
type Widget a b = SF (RawInput, Event WidgetResp, a) (Event WidgetReq, b)
```

`RawInput` is synonymous with Fruit's `GUIInput`, and is added to the incoming signal for the benefit of user-defined widgets. For graphical output, WxFruit provides a `wxpicture` widget (based on WxHaskell's 2D drawing library), which takes 2D drawing primitives as requests.

2.6.6 WxFroot

The design of WxFruit requires the `WidgetResp` and `WidgetReq` types to be a union of all the responses and requests handled by all supported widgets; this is an approach which clearly does not scale well. Additionally, layout containers must route their children's streams, mixing flow of control with the graphical layout. These, along with other issues, motivated the design of WxFroot [**40**].

WxFroot tightly integrates WxHaskell by modifying Yampa so that signal functions can track a widget handle. Widgets are created with a signal function that takes a list of WxHaskell attributes, and yields a proxy that holds the handle to the new widget.

No code has been published for WxFroot, but the referenced paper discusses how it might be implemented. See also Juicy Fruit [**41**] for a similar solution, without modifying Yampa.

2.6.7 Reactive

Reactive [**42**] is a reimplementation of Fran, replacing many bespoke functions with standard type classes, and a new model for reactive behaviours. In addition to Arrows, the library can be used with Applicative Functors and Monads; see the next section on Phooey for an example of this.

2.6.7.1 *Push-Pull Evaluation*

Previous FRP implementations used demand-driven (*pull*) evaluation of behaviours. This would seem to be the only option, as behaviours may vary continuously with time. This is also the native style of functional programming, where the behaviour is expressed as a function of time. However, there are some severe downsides to this approach. Many behaviours are a constant function of time, changing only in response to events. Continuously evaluating these is a waste of resources. Furthermore, the effect of a behaviour-changing event will only occur after the next evaluation, while increasing the polling frequency to reduce latency will have a negative impact on system resources.

Reactive therefore decomposes behaviours into two components: *reactive values* which are independent of time, and *time functions* which are independent of events.

2.6.7.1.1 Reactive Value

A reactive value is constant with regards to time, but changes in response to an event. (A discrete function of time is treated as a reactive value, reacting to timing events.) Reactive values do not require continuous sampling; data-driven evaluation (*push*) is the natural implementation, and there is no latency.

2.6.7.1.2 Time Function

A time function is a continuous function of time, but independent of events. Most time-function behaviours are constants (e.g. a label displays "nothing happening" until something happens), which do not require continuous sampling. In the case of a non-constant function, the behaviour is encapsulated in a polymorphic `time -> value` data type, yielding opportunities for runtime optimisations.

Combinators are used to create hybrid behaviours. A combination of push-pull evaluation ensures that hybrid reactive behaviours are evaluated efficiently and without undue latency.

Many other innovations, including *future values* and *improving values* [**43**] [**44**], allow for a highly efficient implementation.

2.6.8 Phooey

Phooey [**45**] is a multi-paradigm GUI library which uses Reactive. In addition to the traditional Arrow approach, Phooey offers Applicative and Monad interfaces, which are simpler to work with; the Arrow implementation has been dropped altogether from recent releases.

Phooey widgets live in a `UI` monad. Simple input and output widget types are defined as functions between values and `UI`:

```
type IWidget a = a -> UI (Source a)
type OWidget a = Source a -> UI ()
```

(`Source` is a type synonym for reactive behaviours.) Simple examples of `IWidget` and `OWidget` are sliders, which output a value within a fixed range, and a label, which displays its input.

```
showDisplay $ islider (0,10) 5
```

creates a slider (`islider`) with a range of [0,10], and with an initial value of 5. A label (`showDisplay`) will continuously update to show the value of the slider.

2.6.9 Grapefruit

Grapefruit, an FRP framework with a focus on user interfaces, introduces some refinements to existing push-based FRP.

2.6.9.1 *Data Flow System*

Grapefruit incorporates a push-based FRP library [**46**] [**47**], which adds some features not found in Reactive, such as the `merge` function. `merge` combines two signals into a new signal, such that each value in the merged signal is a function of the constituent signals; this is a non-trivial operation in other FRP systems. Grapefruit's data flow system is built up from arrows of type `Circuit i o`, where `i` and `o` are (typically tuples of) signals that form the circuit's input and output respectively.

2.6.9.2 *Grapefruit*

Grapefruit [**48**] is a toolkit agnostic GUI library; the capabilities of any particular toolkit are specified by its type. (Only a small subset of the GTK+ backend has actually been implemented.) In addition to the aforementioned data flow system, Grapefruit comes with an advanced record library for managing signals and widget attributes.

2.6.10 Reactive Banana

Finally, we review Reactive Banana [**49**], one of the newest FRP frameworks.

Reactive Banana is a simple push-pull framework. In addition to the typical time-valued behaviours and events, Banana supports *discrete* behaviours. A `Discrete` value is a function of time, and thus a behaviour. Unlike regular behaviours, however, a discrete behaviour issues an event every time it changes, and cannot vary continuously. This yields significant performance benefits, as listeners can subscribe to the `Discrete`'s event stream, and do not have to continuously monitor a behaviour for changes. Reactive Behaviours (as described in 2.6.2.4) are not supported, thus eliminating a potential time leak when working with accumulating behaviours[2].

A drawback of all previous FRP frameworks is that an adaptor mush be written for each event and widget which the application uses, a tedious manual process. Reactive Banana eliminates this tedium by providing an adapter to WxHaskell, which automatically converts WxHaskell events into FRP events.

2.6.10.1 *The Game*

The first part of Banana.hs (Appendix A, 7.7)'s `main` equation is much the same as the toolkit versions; graphical widgets for the game are created and arranged on a form. However, no event handlers are attached; this is handled by the FRP network, introduced with the `compile` function.

The game's state is represented by the output of `OX.move`, `(Game, Maybe Token)`, where `Game` is the state of the board, and `Maybe Token` indicates which player has won. This has been given the type synonym `State` for convenience.

The network monad's first task is to import the relevant WxHaskell events as FRP events; this is done with the `event0` method. Then, four reactive values are created, which form the core of the game's functionality:

- The events emitted by `event0` are of type `Event ()`, carrying no information. `moves` replaces these with events that modify the game's state by playing the appropriate square, having type `Event (State -> State)`. These are then merged into a single event stream by the `union` combinator.
- The state of the game is tracked by a `Discrete State`, appropriately called `state`. This is an accumulation (`accumD`) of all the `State -> State` events, beginning with a `newGame`.
- A `Discrete String`, `player`, indicates which player's turn is next. This is extracted from the `state`.
- When a player takes a turn, the correct token must be placed in the square. For this purpose, a `Discrete String` is attached to each widget event; when the widget is activated, the discrete value is updated with the current player's token. These values are held in the list `tokens`.

The `sink` function binds a WxHaskell property to a `Discrete`; this is used to provide visual feedback of the network's state. The squares' `text` properties are bound to `tokens`, displaying the correct

[2] An accumulating behaviour is a function of all the previous events in an event stream. For example, if events are numeric, their aggregated total would be an accumulating behaviour.

token after a move. A similar binding to `enabled` disables the square after it has been filled. A final `sink` to the label control displays `player`.

The `reactimate` function executes arbitrary `IO` actions in response to events; it is used here to handle the game's end. `changes` fires an event whenever `state` changes, and these events are filtered (`filterE`) to find the end-game event (i.e. when the `Maybe Token` component holds a value).

Previous examples used existential types to work with the different types of widget in a generic list. This is not possible for Reactive Banana, as the event handlers are attached by the network, which must know the widget's type. Instead, WxHrskell's `objectCast` is used to cast all the widgets to a generic `Control`.

Reactive Banana uses the same WxWidgets toolkit as WxHaskell, and the compiled game is indistinguishable at run time.

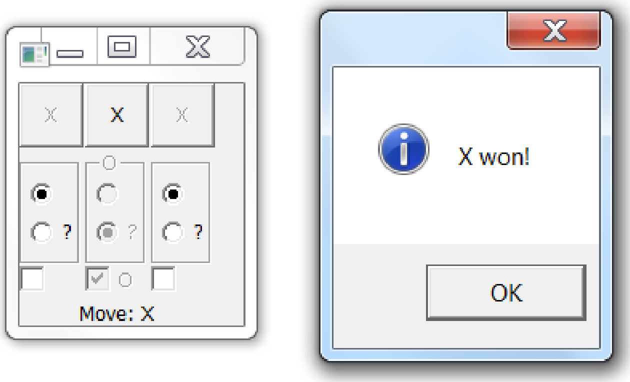

Figure 6: Reactive Banana

3 Analysis and Design

An analysis of the existing work on Haskell GUIs, studying both their successes and failures, leads towards the requirements for a robust, useable, functional GUI library. In the following Background Analysis, the main points of note from the preceding chapter are highlighted and discussed.

3.1 Background Analysis

There are two approaches to developing Graphical User Interfaces in Haskell:

1. Imperative libraries, which translate the C/++ API of a standard toolkit into Haskell IO actions. This is not a desirable state of affairs, as the Haskell programmer prefers to work with higher abstractions; IO actions should only be used to glue these abstractions together, or to perform a small handful of operations.
2. Libraries utilising the principles of Functional Reactive Programming, whereby GUI elements are connected together in a declarative, functional fashion, forming a network of signal processors. This is a much more attractive approach, as it describes an abstraction of GUIs, rather than specifying a series of actions to assemble them.

While the second approach would appear to be far preferable, it would be difficult to describe the work in this direction as universally successful. Few of the many attempts at applying FRP to GUIs have resulted in a fully functional library. On the rare occasion that such a library is achieved, it is uniformly ignored by application developers, who invariably utilise one of the imperative toolkits. It behoves us to investigate why this is the case.

3.1.1 Abstractions for User Interfaces

FRP, with its network of signals and processors, is closely related to control engineering; indeed, the popular Yampa [**34**] library was developed as part of a robot control system. This is also an effective abstraction for arcade games, which typically consist of many interacting graphical elements, hence the popularity of arcade games as proofs of concept for FRP libraries (e.g. [**36**]). Unfortunately, arcade games are a very specific subset of graphical applications. The aim of this project will be to develop a GUI library appropriate for general widget-based applications, e.g. line-of-business systems.

3.2 Requirements Specification

An overview of the requirements for typical GUI applications will lead to the library's functional requirements.

3.2.1 Overview

The functional portion of the user interface for a typical data processing application consists of two kinds of widget:

Command Widgets

Command widgets, such as buttons, perform some function when activated. This is achieved by binding a callback to the appropriate event.

Data Bound Widgets

Many widgets are bound to the state of the application, or some other data source. For example, an indicator may change colour to signal whether the application is online, while a text box may be bound to the contents of a database field. In this case, the binding is two-way; changes to the data source are reflected by new text in the text box, while editing the text will update the database.

3.2.2 Data Binding

A key technology of Rapid Application Development (also known as fourth generation programming) is data binding. With a few mouse clicks and drags, a Delphi or .NET software developer can build a working application that browses and edits a database. This is all achieved by the IDE's ability to write the boilerplate application code, and configure the data binding. While building a RAD IDE for Haskell is outside the scope of this project, a data binding framework should not be too challenging.

Data binding could certainly be implemented with FRP. Changes to the data would send a signal to the bound widget, which then updates its representation. Similarly, changes to the widget's value would send a signal to the data source, if the binding is two-way. However, this may not be the easiest approach. FRP frameworks are large, complex endeavours. They are designed to manage complex networks of signal processors, and to efficiently handle continuous signals. Furthermore, FRP frameworks have no special facilities for formatting and parsing signals, the basic data binding operation. While this could be implemented as custom signal processors, a framework specifically targeted at data binding would be much simpler both in its API and internal implementation.

3.2.2.1 *Back to the IO Monad*

Most graphical user interfaces are inherently stateful and imperative (this is discussed further in 4.1.1). FRP exploits specific domains where this is not the case; otherwise, the IO Monad is as well suited to user interfaces as it is to any other IO work. Rather than seeking a high-level abstraction to replace all imperative GUI programming, a more pragmatic approach than would be to identify common scenarios, and then find a way to make these easier to work with, without reinventing the rest of the GUI. This is the approach taken by WxGeneric, and it is the approach which will guide us in data binding.

3.2.3 Functional Requirements

A one-way data binding creates a one-way relationship between two objects; the binding source and the target. Some property of the source is defined as its value, and whenever the value changes, the target is updated to reflect this change. A two-way binding is a pair of one-way bindings, where the target of one binding is also the source of the second; any changes in the target cause a corresponding change in the source. It is important to define two-way binding in such a way that the two bindings do not form a feedback loop.

To make the data binding more useful, this project will focus on binding mutable data to GUI widgets, other forms of binding being relatively rare. In the case of two-way binding, only user changes to the widget are pushed to the source, so there are no feedback loops.

To simplify the implementation, a widget cannot be the target of more than one two-way binding. If such a facility is required, the widget can be wrapped in a custom binding source, which subscribes to the widget's change events.

3.2.3.1 *One-Way Data Binding*

1. A one-way data binding binds a data source to a target.
2. When the value of the data source changes, the target will be automatically updated.

3. The binding specifies a means for computing the source's value.
4. The binding specifies how this value will be represented by the target.
5. Any number of targets can be bound to a single source.

3.2.3.2 *Two-Way Data Binding*

6. A two-way data binding can be created between a binding source and a widget. In addition to the features of a one-way binding, the source will be updated to reflect user interaction with the widget.
7. There must be a specified means for computing the widget's value, and some conversion may be necessary before applying it to the source.

3.2.3.3 *Binding Lists*

8. Data binding scenarios frequently involve lists of data, e.g. a set of records from a database. The library should provide an interface for binding to such a list of data.
9. Arbitrary positions in the list can be sought to. Whenever the binding source is moved to a different position in the list, the bound widgets are automatically updated.

3.2.4 Non-Functional Requirements

1. The API should be simple to use.
2. The framework should be modular and extensible.
3. The binding framework should not require the programmer to learn any new techniques, beyond those normally required for simple GUI development.

3.2.5 Data Analysis

The data types required to implement the requirements are:

3.2.5.1 *Data Source*

The data source can be any mutable value, such as a record in a database. It has type

```
Source v a
```

where v is the type of the containing variable (e.g. `IORef`), and a is the type of the data (e.g. `String`).

3.2.5.2 *Binding List*

A binding list is a list of mutable values, and shares an interface with simple data sources. For example, a set of records from a database forms a list, each element of which is an independent data source. The type, `Source v a`, is the same as that of a data source.

3.2.5.3 *Target*

The target of a binding is a graphical widget, such as a text box, which is updated whenever the data source changes. The target can be of any type, and is not defined by the framework.

3.2.5.4 *Binding*

A data binding requires four pieces of information:

1. The data source.

2. A function to extract the required data from the source. There is considerable flexibility here, as the extractor can specify a property of the source (e.g. a field in a database record), format the data, or any other arbitrary source to data mapping.
3. The binding target.
4. A function to apply the bound data to the target. This function has similar flexibility, in that it could select any property of the target (e.g. the text property of a label), format the data, or any other data to target mapping.

A binding has the type

```
Binding (a -> d) t (t -> d -> IO ())
```

where `a` is the type of the bound data, and `t` is the type of the target. `(a -> d)` is the function which converts the data source into some type `d`; `(t -> d -> IO ())` is a function which applies `d` to the target.

3.2.5.5 *Source Updates*

The binding target(s) must be updated, according to the binding's specification, whenever the data source changes. The update function has the type

```
Source v a -> IO ()
```

as all binding is presumed to take place in the `IO` monad.

3.2.5.6 *Target Updates*

If the binding is two-way, the source must be updated whenever the target changes. This task must be carried out by the target, and as the design is target-agnostic, the type of this function cannot be determined for the general case. However, we do know that it will have an argument

```
(a -> d -> a)
```

i.e. a function for applying new data (`d`) from the target to the source (`a`). The original data is required for situations such as updating a record field; the other fields of the existing data must be maintained intact.

3.3 Design

The typical object-orientated implementation of data binding is not relevant to Haskell, a purely functional language. Nevertheless, a simple functional design can be derived which satisfies all of the requirements.

3.3.1 Overview

Rather than reinventing the wheel and trying to make it do everything, a pragmatic design will make use of existing design patterns, and accept reasonable limitations of scope.

3.3.1.1 *The Model View Presenter Pattern*

The Model View Presenter (MVP) pattern [**50**] was developed my Taligent (a subsidiary of IBM) for their CommonPoint application system, as an evolution of the venerable Model View Controller (MVC) pattern [**51**].

Model

The model of MVP is identical to the model of MVC; it is the application's underlying data store.

View

The application's GUI is its view. In addition to the passive responsibilities of an MVC view, an MVP view can send updates directly back to the model.

Presenter

The presenter is a derived from the model data, transforming it into the format required by view. When the model's data changes, it informs the presenter, which in turn updates the view with a new representation.

For a simple example, consider a list of Forenames and Surnames, and a screen which must display each name in a "Surname; Forename" format. The model is then the list of names, the presenter is a string function transforming each pair of names into the required format, and the view is a label widget.

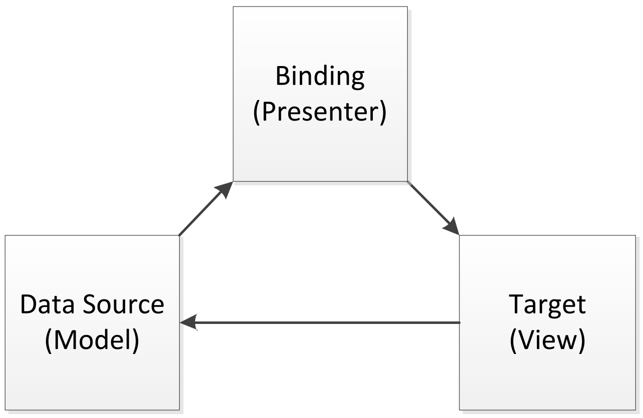

Figure 7: Model View Presenter for Data Binding

The data binding framework is based on MVP, as illustrated in Figure 7. The model is a data source, the binding target is a view, and each binding is a presenter.

3.3.1.2 *The Observer Pattern*

The Observer Pattern is a standard design pattern for components that that react to changes in other components. The subject of the observation maintains a list of observers, and notifies them all when a change occurs.

The data binding framework utilises this pattern for updates. Binding sources, the subjects, encapsulate the data source, and maintain a list of bindings (observers). The data source may only be changed through the binding source's `Variable` interface. When such a change occurs, all the bindings are notified.

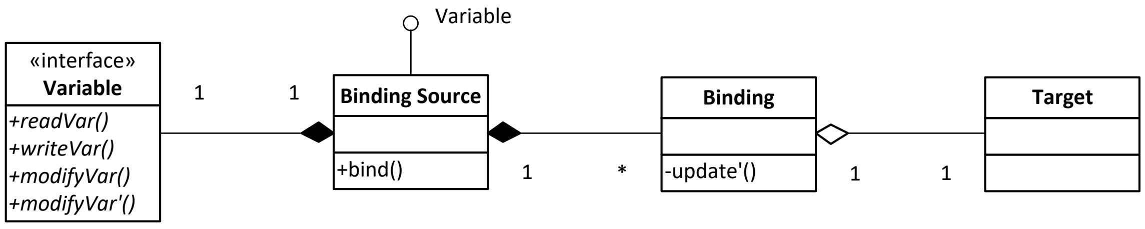

Figure 8: Observer Pattern for Data Binding

Figure 8 shows a UML Class Diagram of how this pattern has been implemented. (There is no standard graphical notation to visualise the structure of functional programs, hence the abuse of UML for this purpose. Haskell's classes are the equivalent of UML's interfaces, while a Haskell data type is modelled as a UML class.)

3.3.1.3 *Two-Way Binding*

Wrapping the target widget in a binding source, and subscribing to the widget's change event, would enable two-way bindings to be created as a complementary pair of one-way bindings. However, as we have accepted a restriction that a widget cannot participate in more than one two-way binding, the second binding source can be dispensed with. Instead, a callback to update the data source is connected directly to the widget's relevant event.

3.3.2 Components

As a functional language, the components of a Haskell program consist of data structures, and functions. The following components are illustrated with simplified excerpts of the full source, given in Appendix B (8.1).

3.3.2.1 *Mutable Variables*

The standard Haskell libraries contain a number of mutable variable types, suitable for different scenarios. As they do not have a standard interface, `Variable` declares a common type class. All operations on these variables will take place in the `IO` monad, along with other GUI code.

```
class Variable v where
    newVar   :: a -> IO (v a)
    readVar  :: v a -> IO a
    writeVar :: v a -> a -> IO ()
```

3.3.2.2 *Binding Source*

A binding source contains two mutable variables. The source's collection of bindings must be mutable, otherwise adding a binding would create two copies of the source; one with the old collection, and one with the new binding. The second variable is the data source, which can only be manipulated through the source's interface. Whenever the data is changed, all the bindings are updated.

The binding source is itself a variable; the source's data is accessed in the same manner as if it was any other variable.

```
data Source v a = Source {bindings :: v [Binding a]
                         ,var      :: v a}
```

3.3.2.3 *Binding List*

A binding list is conceptually similar to a binding source, except that a mutable list of data is encapsulated instead of a single item. While the binding list could be thus implemented, this would be a wasteful approach; the binding mechanism already implemented for simple sources would have to be reimplemented for binding lists.

One approach would be to separate the binding mechanism from the source. However, there is a simpler way. The binding list is composed of a binding source, a mutable list, and a pointer into the list. All binding operations are redirected to the source, and the source's encapsulated data is copied to and from the list as required.

The mutable list is a list of variables, and this list is itself held in a variable. The outer variable enables elements to be added to or removed from the list.

In the same way that the simple binding source does not allow direct access to its data, the binding list does not allow direct access to its list. A number of functions are provided for moving the pointer within the list, and the binding list's `Variable` interface can be used to access the current element. Additionally, there are functions to insert and delete elements.

```
data BindingList v a = BindingList {source :: Source v a
                                   ,list   :: v [v a]
                                   ,pos    :: v Int}
```

3.3.2.4 *Binding Interface*

The procedure for binding simple binding sources and binding lists is identical. Thus, the bind operation is declared as a `Bindable` interface, implemented by both types.

```
class Bindable b where
    bind :: b a -> Binding -> IO ()
```

3.3.2.5 *Binding*

A binding is a 3-tuple of

1. A function which extracts the desired data from the source
2. The binding target
3. A function which applies the extracted data to the target

```
data Binding a = Binding (a -> d) t (t -> d -> IO ())
```

3.3.2.6 *Updating*

The update functions apply data bindings to a binding source's data. They are called by the binding source when it's encapsulated data changes, and are not exported from the module.

```
update' :: a -> Binding a -> IO ()
update' source (Binding extract target apply) =
    apply target $ extract source
```

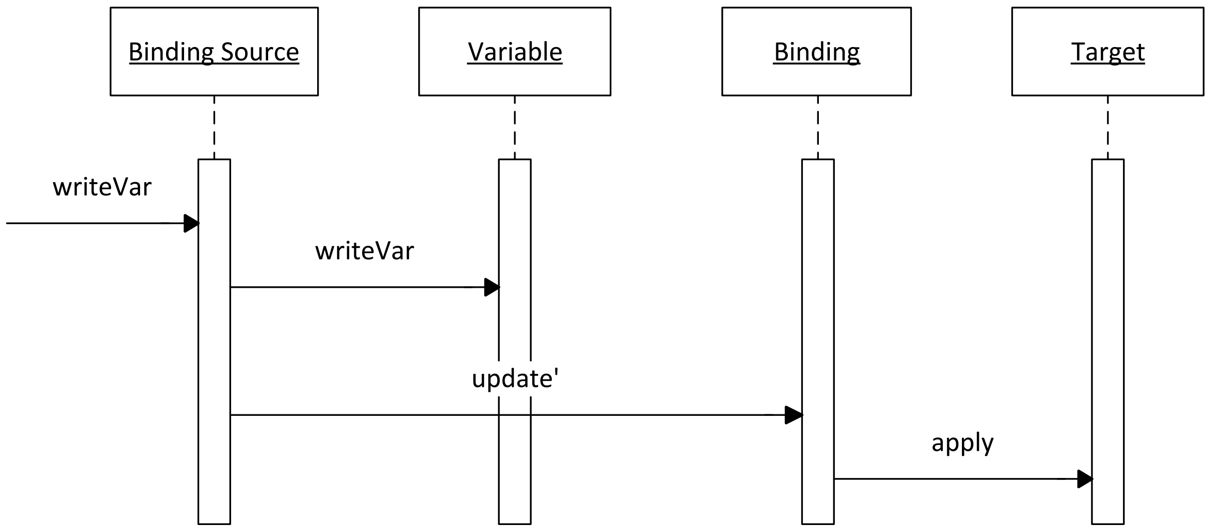

Figure 9: Sequence Diagram of Updating

Figure 9 demonstrates the sequence of updating a binding source. The application calls `writeVar` on the binding source's `Variable` interface; the source forwards this call to its data source. The source then calls `update'` on each binding, which in turn executes its *apply* function on the target.

3.3.2.7 *Binding to Controls*

Graphical toolkits have standard mechanisms for manipulating control properties. These are utilised in convenience functions which create a binding from a source to a control's attribute; the function knows how to read from and write to the attribute. An additional convenience function is provided for textual controls, where only the control needs to be specified, and it is assumed that the control's text attribute should be synchronised with the data source.

There is also a function to create a group of navigation widgets for binding lists. The current position is bound to a spin control, and standard navigation buttons are provided, in addition to insertions and deletions. (See 4.4.2 for an example of this.)

3.3.3 Packaging

The core functionality is toolkit agnostic, and forms the `binding-core` library. Widget-aware convenience functions are packaged separately; `binding-gtk` for the Gtk2Hs interface, and `binding-wx` for the WxHaskell interface.

All the source code is listed in 8Appendix B: Data Binding Source Code. `binding-core` contains three source files:

- `Variable.hs`, which defines the interface for mutable variables
- `Core.hs`, which implements simple data binding
- `List.hs`, which implements binding lists.

`binding-gtk` and `binding-wx` provide `Gtk.hs` and `Wx.hs` respectively, which provide utility functions for binding to graphical widgets.

4 Implementation and Testing

Various issues were encountered and solved during the project's execution; some were of a technical nature, but more severely, it became clear that the project had begun in the wrong direction. In this chapter, these difficulties and their solutions are described. We then proceed to describe the testing strategy adopted to ensure that the data binding code performs as expected.

4.1 Problems and Solutions

A variety of technical obstacles were encountered during the literature review and development phase. However, the most significant issue encountered during the project's execution was that the search for a solution began with an incorrect assumption.

4.1.1 Barking up the Wrong Tree

The project was begun with an assumption that GUI programming in the IO monad is "particularly challenging", and that an alternative abstraction is needed. This led to a review of Functional Reactive Programming, a functional abstraction for graphical user interfaces.

Unfortunately, it was generally not possible to exercise the reviewed FRP frameworks with our noughts and crosses example, as the majority of such frameworks are abandoned projects, or unimplemented research papers. Furthermore, the implementation of this simple game in Reactive Banana demonstrates that is not an expedient approach.

An individual square knows whether it contains nothing, a nought, or a cross. It has no knowledge of the game's global state; thus, a reactive value must be created, holding the game's state. The event stream of each square is attached to this value. After a player takes their turn, the square fires a turn event, and the game state is updated. Further reactive values track the visual state of each square, and whether this turn has ended the game. An event stream is also needed in the reverse direction, from the square states to each square's widget, in order to place the correct token and disable further interaction.

Comparing this to the other code in Appendix A: Noughts and Crosses Source Code we see that the use of FRP has yielded two significant changes to the game's design:

- In the toolkit (`IO` monad) design, each cell fires an event when it is activated. This event reads the current state of the game, displays the appropriate token in the activated cell, and determines whether an end-game state has been achieved. In the FRP design, displaying the correct token is carried out by a separate event, from the game state back to the cell.
- The state of the game is no longer kept in a mutable variable. Instead, several reactive values are used to keep track of various states.

In other words, the use of FRP for a simple board game has doubled the number of events which must be implemented, and replaced a single mutable variable with multiple reactive values, but has not brought any apparent advantages. This is not specific to board games; FRP models the application as a set of interacting (graphical) agents, where each agent determines its own behaviour, in response to events received from the environment or other agents. This is fine for applications which can be thus modelled, but does not work terribly well for programs that work with a global state.

It would seem that the initial quest for a new functional abstraction that would replace the IO monad for all GUI work was a misguided approach. Interacting with the user through a graphical terminal is as much an IO activity as interacting with the file system, and Haskell's IO monad is the natural habitat of IO activity. Of course, abstractions to make the task easier would be appreciated, but these should not attempt to entirely supplant IO actions. Instead, abstractions can be developed for common classes of UI tasks, which will fit into the IO monad along with other GUI work.

4.1.1.1 *The Right Tree*

As noted in the Background Analysis, Functional Reactive Programming is an excellent abstraction for arcade-style computer games. It is also well suited to multimedia applications; much of the work on FRP was carried out in the context of computer-driven animations. However, none of this is relevant to typical line-of-business desktop applications.

A common requirement for business applications is to bind graphical widgets to a data store. For example, a form for editing personnel data may be bound to a personnel database; the operator is then able to modify the database by editing the content of the form's graphical fields. Data binding is a fundamental capability of rapid application development, but there are currently no Haskell libraries to abstract the common elements of this task. Hence, we chose to pursue the more pragmatic task of developing a data binding framework, and abandoned the quest for one GUI abstraction to rule them all.

4.1.2 Technical Issues

Haskell is primarily an academic language, with some niche industry applications. One consequence of this is that, unlike languages which are widely used for general purpose programming, there is no commercial ecosystem, and open-source attention is relatively modest. Commercial operating systems are not supported to the same extent as their free competitors, and there is a dearth of development tools. Furthermore, the finesse and quality which may be imposed on commercial products are often lacking in Haskell's offerings.

4.1.2.1 *Integrated Development Environment*

When work began on the project, there were two actively developed IDE's for Haskell; Leksah [**52**] and EclipseFP [**53**]. Leksah was found at the time to be highly problematic on Windows, suffering from numerous bugs and an unwieldy interface. EclipseFP, on the other hand, is an Eclipse plug-in, rather than a standalone program, and as such benefits from Eclipse's considerable polish. However, work on integrating Eclipse with GHC had run into considerable difficulties [**54**], and EclipseFP was of little use.

The overwhelming majority of Haskell programmers [**55**] use a generic programmer's editor, and this was the solution initially resorted to. While workable for very small projects, this became increasingly inconvenient as the project grew. Management of the project's resources and build configuration all have to be done by hand, as the editor does not understand Cabal files. IDEs such as Visual Studio and Eclipse (for better supported languages) provide numerous features for understanding and working with code, which are all absent or poorly supported in generic editors. (Emacs can provide some of these facilities through plug-ins, if the user is comfortable with much manual configuration and memorising dozens of obscure hotkeys.)

EclipseFP version 2 was released last December, finally bringing a useable cross-platform Haskell IDE. Although very spartan by modern IDE standards, EclipseFP understands how to structure and configure a Haskell project, and provides basic coding facilities such as continuous background compilation, and limited code insight. Hopefully the project will continue to provide a high quality IDE for Haskell.

4.1.2.2 *Windows*

The Haskell ecosystem is Linux centric. Although Windows users are not neglected, problems still tend to arise. For example, I was unable to use Leksah (it may have been expecting standard *nix file paths), and Glade complained about missing C libraries. Building libraries with non-Haskell dependencies is often much more complicated on Windows, and every package seems to require a different version of MinGW/MSYS/Perl.

In retrospect, it may have been easier to develop on a Linux virtual machine. Nevertheless, there is usually adequate documentation to get the job done eventually, and the community is helpful to those who cannot manage alone.

4.1.2.3 *Gtk2Hs and WxHaskell*

Gtk2Hs and WxHaskell, with their dependencies on C/C++ libraries, suffer from the usual problems associated with installing on Windows. In addition to this, Gtk2Hs ships with a broken setup script, and as a consequence, does not install on the current version of the Haskell Platform without significant tweaking [**56**]. Furthermore, the code repositories of both projects were recently missing for several months, along with Gtk2Hs's web site, and both projects are built against older versions of their underlying toolkits.

Generating useable language bindings for such a large library is a great deal of work. This is particularly difficult for WxHaskell, as Haskell lacks a foreign function interface to C++; even with a great deal of manual effort, WxHaskell only covert a subset of WxWidget's functionality.

4.2 Testing

The two types of testing relevant to this project are unit testing and integration testing. Acceptance testing, performed by stakeholders or other users, is not applicable to a new development that has no users; neither is regression testing applicable to this first version of the package.

Cabal, described in 2.3.2, has functionality (`cabal test`) to automatically execute test suites included with a package and log the results.

4.3 Unit Testing

A *unit test* verifies that a function gives the correct result for a given input. Multiple tests cases may be provided for a single function to test across a broad range of inputs, particularly edge cases. Unit tests are assembled into test suites, which aim to cover every function exported by the module or package under test.

The `binding-gtk` and `binding-wx` packages are thin GUI wrappers over the `binding-core` API, which add no significant functionality. As such, it was felt that unit testing efforts should be concentrated on the core `binding-core` package.

Unit testing is normally performed with the aid of a framework for automatically running test scripts, performing ancillary operations, and reporting the test results. The most popular testing frameworks for Haskell are HUnit [**57**] and QuickCheck [**58**].

4.3.1 HUnit

HUnit is a member of the xUnit family of unit testing frameworks, modelled after the venerable JUnit for Java. Each test exercises one function, and compares the *actual* return value to the *expected* result. This comparison is performed by an assertion, which determines whether the test has passed or failed. Various utility functions are provided to build assertions, assemble test suites, and report the results.

The HUnit tests for `binding-core` are given in `tests\HUnit.hs` in Appendix B. Random values are used to construct test data, thereby ensuring a greater range than if the data was hard coded.

```
Cases: 13  Tried: 13  Errors: 0  Failures: 0
```

Figure 10: HUnit test results

4.3.2 QuickCheck

QuickCheck offers a more declarative approach to testing than traditional unit testing frameworks. Instead of a series of imperative operations, a QuickCheck test consists of a (generally) pure *property*, which evaluates to a Boolean. For example, we can test whether doubling a number always returns an even result with

```
prop_even :: Int -> Bool
prop_even x = even (x * 2)
```

Given this function, QuickCheck will automatically create a set of test data (integers in this simple example), evaluate the function for each, and print the result. When generating test data, QuickCheck will try to insure that the data covers a good range, including corner cases. If a test fails, QuickCheck will try to reduce the test case to the simplest form that fails, e.g. the shortest list or smallest number. To run a test suite, QuickCheck provides the `quickCheckAll` function, which uses Template Haskell to run all the tests in its module.

4.3.2.1 *Modifiers*

QuickCheck provides modifiers for restricting test data. For example, if we wanted to limit the above test to positive numbers, we could write

```
prop_even :: Positive Int -> Bool
prop_even (Positive x) = even (x * 2)
```

4.3.2.2 *Implications*

If the built-in modifiers are not sufficient, the simplest solution is to write an *implication*, such as

```
prop_even :: Int -> Bool
prop_even x = x > 0 ==> even (x * 2)
```

This example will discard all non-positive test data. An important difference between modifiers and implications is that modifiers are a distinct type. In this example, `Positive a` is a new type defined by QuickCheck, for which only positive numbers are generated. However, when using the `x > 0` implication, QuickCheck generates test data over the full range of `Int`; test cases failing to satisfy the condition are then discarded.

4.3.2.3 *Arbitrary*

QuickCheck knows how to generate test data for all types of the class `Arbitrary`. `Arbitrary` instances are provided for many standard Haskell types, but if the programmer has defined their own types, they must provide their own generator. `Arbitrary` has two methods; `arbitrary` and `shrink`:

- `arbitrary` is the method called by QuickCheck when generating test data. `arbitrary` operates in the `Gen` monad, which provides a rich set of combinators for building new test generators out of existing generators.
- `shrink`, which does not have to be defined, is used to reduce failing test cases to their simplest form.

The functions in `Binding.List` are only meaningful on lists of a minimum size. Non-empty lists can be easily generated with the `NonEmpty` modifier, but many operations, such as deletion, require at least two elements. This was initially implemented with an implication `length xs > 1`.

Unfortunately, QuickCheck's list generator is biased towards short lists, with the result that much time is wasted as many test cases are generated but not used. Furthermore, the test cases are still biased towards short lists; a better test set would use a broader range of lists sizes.

The solution is to declare a newtype for the type of list that we wish to generate. The snippet

```
newtype List = List [A] deriving Show

instance Arbitrary List where
    arbitrary = liftM List $ choose (2, 100) >>= vector
    shrink (List xs) = [List ys | ys <- shrink xs, P.length ys > 1]
```

declares a type List, based on the standard list type. The Arbitrary instance for this type creates lists that have a random length between 2 and 100. We have also implemented shrink, which enables QuickCheck to reduce failing test cases to the simplest failing case. shrink for List is a simple wrapper over shrink for regular lists, discarding any results which are too small.

4.3.2.4 *Position Parameter*

Most Binding.List functions take an Int parameter, which points to a position (or offset) in the list. While QuickCheck will generate test data across the full domain of Int, only valid positions in the list are suitable for testing. The simple solution is to restrict the generated test data with an implication. However, so many test cases are thereby discarded that, in addition to the time penalty noted above, QuickCheck will give up with a message such as

```
*** Gave up! Passed only 9 tests.
```

Custom Arbitrary types won't help here, because the definition of a valid position depends on the size of the generated list, which isn't known until run time.

To get around this, we have defined a helper function

```
-- | Maps @i@ to a position in @xs@.
anywhere :: Int -> [A] -> Int
anywhere i xs = let max = P.length xs - 1
                in if max == 0 then 0 else i `mod` max
```

Given a random number and list, anywhere will use modular division to generate a random position within the list. notLast operates similarly, but excludes the end of the list.

4.3.2.5 *Testing IO Actions*

A regular QuickCheck Property evaluates to Bool. This creates a problem for testing IO actions, as their result cannot be extracted from the IO monad to yield a pure Bool. For this purpose, QuickCheck defines the PropertyM monad for testing monadic properties. Within PropertyM, the result of an IO action is obtained by run, and this result is then verified by assert. monadicIO is used to transform a PropertyM into a Property.

The details of how QuickCheck tests monadic properties are given in [**59**].

```
=== prop_remove on tests\QuickCheck.hs:44 ===
+++ OK, passed 100 tests.
=== prop_removeLast on tests\QuickCheck.hs:51 ===
+++ OK, passed 100 tests.
=== prop_insert on tests\QuickCheck.hs:57 ===
+++ OK, passed 100 tests.
=== prop_Source on tests\QuickCheck.hs:113 ===
+++ OK, passed 100 tests.
=== prop_Length on tests\QuickCheck.hs:118 ===
+++ OK, passed 100 tests.
=== prop_seek on tests\QuickCheck.hs:123 ===
+++ OK, passed 100 tests.
=== prop_position on tests\QuickCheck.hs:128 ===
+++ OK, passed 100 tests.
=== prop_seekBy on tests\QuickCheck.hs:133 ===
+++ OK, passed 100 tests.
=== prop_next on tests\QuickCheck.hs:141 ===
+++ OK, passed 100 tests.
=== prop_prev on tests\QuickCheck.hs:146 ===
+++ OK, passed 100 tests.
```

Figure 11: QuickCheck test results

4.4 Integration Testing

The binding-gtk and binding-wx packages do not add new functionality; rather, they provide graphical toolkit bindings for the functionality of binding-core. As such, they are good candidates for integration testing.

Whereas a unit test verifies that a function operates correctly in isolation, an integration test verifies that the different components of a system work correctly in aggregate. binding-gtk, for example, is a glue layer between binding-core and Gtk2Hs; an integration test will confirm that these three layers are interoperating correctly.

The integration tests consist of two programs for each package:

4.4.1 Simple Data Binding

simple.hs demonstrates simple data binding; a numeric data source is two-way bound to each of two text boxes. Thus, any changes made to one box will be reflected in the other.

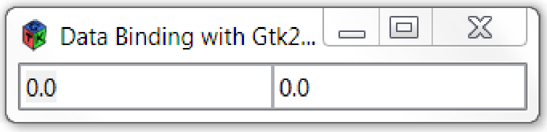

Figure 12: Simple data binding with Gtk2Hs

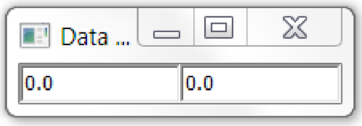

Figure 13: Simple data binding with WxHaskell

4.4.2 Binding Lists

lists.hs demonstrates the more complex scenario of binding to lists of data. When the program starts, it loads a personnel database from in.txt. This contains a list of the simple Person data type, in Show format. A navigation control is created to move through the list, and the individual fields are bound to graphical widgets. When the application exits, the modified database is written to out.txt.

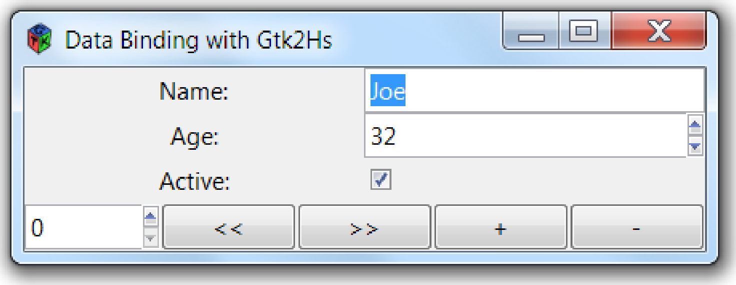

Figure 14: Binding lists with Gtk2Hs

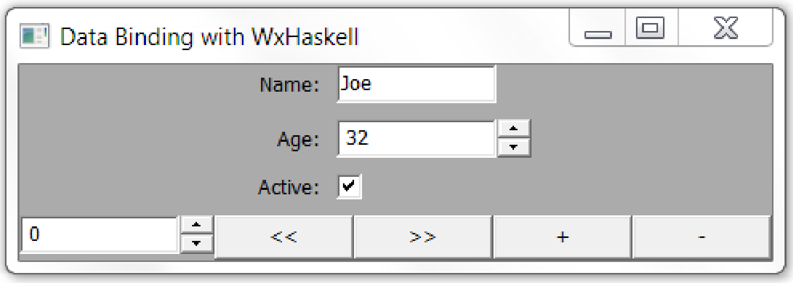

Figure 15: Binding lists with WxHaskell

4.4.3 Test Execution

It is technically feasible to drive integration tests from a unit test framework. For example, after executing `simple.hs`, a unit test might change the value in one text box, and then check whether the other text box has been correctly updated. However, this would be assuming that programmatically firing a particular button click event has the same effect as the user clicking the button, and that the property read by the test is an accurate reflection of the widget's display. This is not an adequate approach, as the correctness of such assumptions in the package under test is itself being tested. Hence, these tests were verified manually. (In a larger project, many integration tests may be programmatically driven to achieve the advantages of automated testing, while additional tests are manually executed.)

4.5 Test Results

All the unit tests pass successfully, and the integration tests perform as expected. However, that the tests pass is not sufficient; it must be shown that the tests cover all of the project's requirements.

4.5.1 Functional Requirements

Each functional requirement must be satisfied by a function in the library's API. To prove that the function operates as intended, it must be tested. The following table lists each functional requirement, its corresponding API function, and the function's tests. These could be HUnit and QuickCheck unit tests, or, for the GUI-oriented requirements, a manual integration test.

	Requirement	API	Tests
1.	A one-way data binding binds a data source to a target.	`Binding.Core.bind`	HUnit: `testSource` QuickCheck: `prop_Source`
2.	When the value of the data source changes, the target will be automatically updated.	The update function called by `Source`'s `writeVar`	HUnit: `testSource` QuickCheck: `prop_Source`
3.	The binding specifies a means for computing the source's value.	`bind`'s second parameter	HUnit: `testSource` QuickCheck: `prop_Source`
4.	The binding specifies how this value will be represented by the target.	`bind`'s fourth parameter	HUnit: `testSource` QuickCheck: `prop_Source`
5.	Any number of targets can be bound to a single source.	Repeated applications of `bind` to a `Source`	`simple.hs` 2 text boxes are bound to the same binding source.
6.	A two-way data binding can be created between a binding source and a widget. In addition to the features of a one-way binding, the source will be updated to reflect user interaction with the widget.	`bindControl` in `Binding.Gtk` and `Binding.Wx`	`simple.hs` & `lists.hs` In both integration tests, widgets are two-way bound to a binding source.
7.	There must be a specified means for computing the widget's value, and some conversion may be necessary before applying it to the source.	`bindFromControl`'s third parameter	`simple.hs` & `lists.hs` The correct attribute of each widget is bound to the source, including some conversion between integers and real numbers.
8.	Data binding scenarios frequently involve lists of data, e.g. a set of records from a database. The library should provide an interface for binding to such a list of data.	`Binding.List.BindingList`	HUnit: `testList` QuickCheck: `prop_List` `lists.hs` A list of `Person` data is bound to a set of widgets.

9. Arbitrary positions in the list can be sought to. Whenever the binding source is moved to a different position in the list, the bound widgets are automatically updated.	`seek`, and other navigation functions	HUnit: `testSeek` QuickCheck: `prop_Seek` Similar tests exist for other navigation functions. `lists.hs` The navigation controls can be used to seek to any position within the list.

4.5.2 Non-Functional Requirements

Unquantified non-functional requirements cannot be formally proven. However, it should be clearly demonstrable that they have been satisfied.

1. The API should be simple to use.	A binding source is created with a single call to `newVar`. One further function call is sufficient to create a one- or two-way binding.
2. The framework should be modular and extensible.	Any mutable reference can be used as a data source, if given as an instance of `Variable`. One- and two-way bindings can be made to any `IO` object, and interfaces such as those provided for Gtk2Hs and WxHaskell can be easily written for any other toolkit.
3. The binding framework should not require the programmer to learn any new techniques, beyond those normally required for simple GUI development.	All of the library's functionality is exposed as regular `IO` actions. Data sources (`Variables`) mimic the interface to existing mutable reference types.

5 Evaluation

The stated aim of this project was "to critically review past efforts at developing GUI functionality for Haskell, and investigate how such functionality can be better supported". The original vision was to develop some new abstraction that would replace imperative GUI programming in its entirety; however, an analysis of existing work clearly demonstrates that no such one-size-fits-all solution exists.

Instead, we have shown that a more pragmatic approach is to retain the existing imperative toolkits, as a basis on which domain-specific abstractions can be built. Past work on Haskell GUI functionality has focused on Functional Reactive Programming, which is an effective solution for the domain of multimedia applications and video games.

The domain of data processing applications, ubiquitous to corporate desktops, has hereunto been ignored. In redress of this neglect, we have developed a data binding framework. This framework targets the common requirement of manipulating business data through a graphical interface, eliminating a significant differentiator between Haskell and "Business Languages" such as C# and Java.

Although the initial vision of one framework to support all GUI programming has not been realised, our analysis of the solution space has indeed fulfilled the project's aim; we have shown the direction that future work should take to enhance Haskell's GUI support. Furthermore, we have made significant progress in the goal of making Haskell a suitable candidate for line-of-business software development.

5.1 The List in Binding Lists

The binding list's data is held in a singly linked list of `Variables`. Two objections may be raised against this choice:

1. Seeking an element in such a list is O(n), whereas seeking into an array is O(1).
2. Haskell already has mutable arrays [**60**]; why reinvent them?

Before answering these objections, it may be worth first presenting the argument in favour of the common list; it is well supported by common libraries, and every functional programmer is intimately familiar with them.

5.1.1 Performance

Concern over the performance of seeking into a binding list is clearly a case of premature optimisation, "the root of all evil" [**61**]. The seek operation is thousands of times faster than repainting the GUI to show the result of seeking, and similar orders of magnitude faster than the human operator who is interacting with said GUI.

In the case of an exceedingly long list, with at least millions of elements, seek time may become an issue. In that case, profiling the binding list's performance and memory usage would be in order, to determine where the actual bottlenecks lie. Without such measurements, assumptions about the effects of changing a particular data structure on real world performance are little more than idle speculation.

5.1.2 Reinventing the `IOArray` Wheel

The standard Haskell libraries provide `IOArray`, a mutable array in the `IO` Monad. This would have absolved us of having to develop our own data structure for binding lists, along with a considerably aggravated risk of functional defects and poor performance (particularly under load). The standard libraries are developed by highly skilled and experienced engineers, and reviewed by a broad range of Haskell users; much more than can be said of our humble development efforts.

The motivation for using a list of `Variables` is simply that this provides a different opportunity for code reuse. The `Variable` interface and `Binding.Core` provide all the functionality needed to implement data binding; binding lists only have to generalise this to a list of data sources. The use of a different data type would have necessitated a considerable amount of plumbing code to translate between `IOArray` operations and `Variable` functions.

Given the choice, reusing `Variable` was chosen over reusing `IOArray`, as this leads to simpler, more consistent code.

6 Summary and Conclusions

A summary of the project's achievements and conclusions leads to some thoughts about the work's potential, and how it could be extended.

6.1 Summary

Modern programming languages may be broadly divided into imperative and functional languages. Almost all commercial software development is done with imperative languages, such as C#, Java, and PHP. In contrast, functional languages such as Haskell and ML are the domain of academia and some specialist commercial niches, such as finance.

Interest in functional languages, particularly Haskell, has been growing outside of their traditional domains. Modern CPUs are increasing the number of cores at a much greater rate than the speed of each individual core, and programs which can run in multiple simultaneous threads stand to benefit from a significant speed boost [**62**]. Traditional imperative programs, modelled on mutable global state, are notoriously difficult to parallelise [**63**]; many software designers are turning to functional languages for a solution, as referential transparency and immutable data eliminate many barriers to multi-threaded programming. Furthermore, Haskell's expressive type system (System F_C [**64**]) can capture many programming errors at compile time; run time errors are much harder to find and diagnose. Thus, there is a need for functional languages to support programming tasks outside of their traditional domains.

This project has focused on one such task, the graphical user interface, with Haskell, a purely functional programming language. All of a Haskell program's interaction with the environment takes place in the `IO` monad, an abstraction which enables IO in the context of lazy evaluation and referential transparency. While this suffices for simple tasks, such as reading a file from disk, maintaining a GUI is a much more complex endeavour, and significantly more support is needed.

Two categories of GUI library for Haskell have been reviewed. Gtk2Hs, WxHaskell, and QtHaskell, all provide Haskell bindings to independently maintained cross-platform graphical toolkits. Although the bindings make use of Haskell language features to present a more comfortable interface, the programming paradigm is, like that of the underlying toolkits, imperative. An alternative approach to supporting GUIs from Haskell is that of Functional Reactive Programming. Implemented in several different frameworks, this takes a more declarative approach, modelling the user interface as a network of graphical widgets and event streams.

Neither of these approaches suffices for common application domains, such as interactive data processing, or any other application which is focused on global state. The toolkit bindings are too low level, and require reams of C-like imperative code to manage a moderately sized interface. FRP assumes that widgets need to interact with the environment and a handful of other widgets; it does not address the need of widgets to interact with global program state. It is clear that new abstractions are needed.

Rapid Application Development environments (i.e. Microsoft Visual Studio and Embarcadero Delphi) support data binding, whereby graphical widgets are bound to the application's data. Editing the widget's bound attribute, such as the text field of a text box, will automatically update the bound data. Similarly, changes to the data are automatically propagated to the bound widget. A data binding framework for Haskell would go a long way towards closing the gap between functional and imperative languages when targeting the corporate desktop.

Thus, we have created a framework for binding mutable Haskell data to a GUI. The framework is toolkit agnostic, and interfaces have been provided to Gtk2Hs and WxHaskell. Both one- and two-way bindings between mutable variables and widgets can be easily created. The framework is provided with sample programs, and thorough unit tests for both HUnit and QuickCheck.

6.2 Conclusions

We have shown that, after reviewing all significant literature on the subject, there is no single abstraction or framework capable of supplanting imperative programming for all GUI development. However, this does not mean that the search for alternatives is futile. Rather, we have demonstrated how Haskell's imperative IO monad is an environment in which domain-specific frameworks can be combined with miscellaneous GUI actions.

Prior work on Functional Reactive Programming focuses on one such domain, that of interacting graphical elements. We have developed a framework to support the domain most commonly required by businesses, that of data binding. Thus, it has been demonstrated that a pure functional language such as Haskell can be quite capable of those tasks commonly considered to be the domain of imperative languages.

6.3 Future Work

There is much potential to expand the core data binding functionality already developed. The next stage would be what Microsoft terms "complex data binding" [**65**]; multiple elements of a binding list are shown simultaneously, such as in a grid. A further feature, which is tricky to get right even with highly mature data binding frameworks, would be for data sources to link together. For example, we may have a list of customers and a list of orders; there is a one-to-many relationship between customers and orders. Selecting a customer from the customers binding list should bind all of the customer's orders to an orders grid.

The other side of the data binding coin is persistence; a data source which knows how to save itself to and load itself from persistent storage would bring the framework closer to modern data binding expectations.

The use of object-relational mapping to support these scenarios in the context of object-orientated programming has been termed "the Vietnam of computer science" [**66**]. Much of the impedance mismatch between object-orientated languages and relational databases stems from inheritance and encapsulation (the bundling of data with behaviour), concepts which do not feature strongly in functional languages. If functional languages had full support for data binding and persistence, would they be *more* suitable for line-of-business applications than object-oriented languages?

Finally, it is worth noting that nothing about the `binding-core` package is specific to graphics or user interfaces. Perhaps there is potential for binding data to different targets, such as synchronising equivalent data held in different stores.

7 Appendix A: Noughts and Crosses Source Code

The source code in this appendix was compiled with the following package versions:

Haskell Platform:	2011.2.0.1
Gtk2Hs:	0.12
WxHaskell:	0.12.1.6
QtHaskell:	1.1.4
WxGeneric:	0.6.1
Reactive Banana:	0.4.1.1

7.1 OX.hs

```haskell
module OX (Square, Token(..), showBoard, Game(..), newGame, move) where

import Data.Array
import Data.List

data Token = None | X | O
    deriving Eq

-- |The coordinates of a square.
type Square = (Int,Int)

-- |A noughts and crosses board.
type Board = Array Square Token

-- |Returns an empty 'Board'.
newBoard :: Board
newBoard = listArray ((1,1),(3,3)) (repeat None)

-- |Puts a 'Token' in a 'Square'.
setSquare :: Board -> Square -> Token -> Board
setSquare board square token =
    if (board ! square) /= None
    then error $ "square " ++ show square ++ " is not empty"
    else board // [(square, token)]

-- |Determine if the 'Board' is in an end state.
--  Returns 'Just' 'Token' if the game has been won, 'Just' 'None' for a draw,
otherwise 'Nothing'.
endGame :: Board -> Maybe Token
endGame board
     | Just X `elem` maybeWins = Just X
     | Just O `elem` maybeWins = Just O
     | None `notElem` elems board = Just None
     | otherwise = Nothing

    where rows :: [[Square]]
          rows = let i = [1..3]
                 in [[(x,y) | y <- i] | x <- i] ++ -- rows
                    [[(y,x) | y <- i] | x <- i] ++ -- coloumns
                    [[(x,x) | x <- i], [(x,4-x) | x <- i]] -- diagonals
          rows2tokens :: [[Token]]
          rows2tokens = map (map (board !)) rows

          isWin :: [Token] -> Maybe Token
          isWin tokens
              | all (==X) tokens = Just X
              | all (==O) tokens = Just O
              | otherwise = Nothing

          maybeWins :: [Maybe Token]
          maybeWins = map isWin rows2tokens

-- |The state of a game, i.e. the player who's turn it is, and the current board.
data Game = Game Token Board

newGame :: Game
newGame = Game X newBoard

-- |Puts the player's token on the specified square.
-- Returns 'Just' 'Token' if the game has been won, 'Just' 'None' for a draw,
otherwise 'Nothing'.
move :: Game -> Square -> (Game, Maybe Token)
move (Game player board) square =
    let board' = setSquare board square player
        player' = case player of {X -> O; O -> X}
    in (Game player' board', endGame board')

-- **************************************** Show instances
*********************************************

outersperse :: a -> [a] -> [a]
outersperse x ys = x : intersperse x ys ++ [x]

instance Show Token where
    show X = "X"
    show O = "O"
    show None = " "
    showList tokens = showString $ outersperse '|' $ concatMap show tokens

-- Board cannot be declared an instance of Show, as this would overlap with the
existing instance for Array
showBoard :: Board -> String
showBoard board =
    let border = " +-+-+-+"
        i = [1..3]
        showRow x = show x ++ show [board ! (y,x) | y <- i]
    in intercalate "\n" $ "  1 2 3" : outersperse border (map showRow i)

instance Show Game where
    show (Game player board) = showBoard board ++ "\n\nTurn: " ++ show player
```

7.2 Console.hs

```
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Exception
import OX

play :: Game -> IO ()
play game = do
            putChar '\n'
            print game
            input <- getLine
            let (game'@(Game _ board), result) = move game $ read $ '(' : input ++
")"
            attempt <- try $ evaluate result
            case attempt of
                Left (error::ErrorCall) -> print error >> play game
                Right result -> case result of
                                    Nothing -> play game'
                                    Just token -> putStrLn $ "\n" ++ showBoard
board ++ "\n" ++ if token == None then "Draw" else show token ++ " won!"

main = play newGame
```

7.3 Gtk2HS.hs

```haskell
import Control.Monad
import Data.IORef
import Graphics.UI.Gtk

import OX

main = do
    --create a new game
    game <- newIORef newGame

    -- create the main window
    initGUI
    window <- windowNew
    set window [windowTitle := "OX"]
    onDestroy window mainQuit

    vbox <- vBoxNew False 0
    set window [containerChild := vbox]

    label <- labelNew $ Just "Move: X"
    boxPackEndDefaults vbox label

    table <- tableNew 3 3 True
    boxPackEndDefaults vbox table

    btns <- replicateM 3 toggleButtonNew

    radios <- replicateM 3 $ do
        group <- vBoxNew False 0

        button <- radioButtonNewWithLabel "" -- radioButtonNew doesn't line up
with radioButtonNewWithLabel
        button `on` toggled $ set button [widgetSensitive := False]

        button_ <- radioButtonNewWithLabel "?"
        set button_ [radioButtonGroup :=  button]

        mapM_ (boxPackStartDefaults group) [button, button_]
        return (button_, group)

    checks <- replicateM 3 checkButtonNew

    --attach event handlers
    let event square button = button `on` toggled $ do
            g@(Game player _) <- readIORef game
            set button [buttonLabel := show player]
            let (g'@(Game player _), result) = move g square
            case result of
                Nothing -> do --continue game
                           writeIORef game g'
                           set button [widgetSensitive := False]
                           set label [labelLabel := "Move: " ++ show player]

                Just token -> do --end game
                              let message = case token of
                                                  X -> "X won!"
```

```
                                                    O -> "O won!"
                                                    None -> "Draw!"
                              messageDialogNew Nothing [] MessageInfo ButtonsOk
message >>= dialogRun
                              widgetDestroy window

    zipWithM_ (\b x -> event (x,1) b >> tableAttachDefaults table b (x-1) x 0 1)
btns [1..3]
    zipWithM_ (\(r,g) x -> event (x,2) r >> tableAttachDefaults table g (x-1) x 1
2) radios [1..3]
    zipWithM_ (\c x -> event (x,3) c >> tableAttachDefaults table c (x-1) x 2 3)
checks [1..3]

    --run the UI
    widgetShowAll window
    mainGUI
```

7.4 WxHaskell.hs

```
import Control.Monad
import Graphics.UI.WX

import OX

main = start $ do
    --create a new game
    game <- varCreate newGame

    -- create the main window
    window <- frame [text := "OX"]
    label <- staticText window [text := "Move: X"]

    btns <- replicateM 3 $ button window [size := sz 40 40]
    radios <- replicateM 3 $ radioBox window Vertical ["", "?"] []
    checks <- replicateM 3 $ checkBox window [text := "  "] --reserve space for
X/O in label

    set window [layout := column 5 [grid 1 1 [map widget btns, map widget radios,
map widget checks], floatCenter $ widget label]]

    --attach event handlers
    let event square btn = do
            g@(Game player _) <- varGet game
            set btn [text := show player]
            let (g'@(Game player _), result) = move g square
            case result of
                Nothing -> do --continue game
                           varSet game g'
                           set btn [enabled := False]
                           set label [text := "Move: " ++ show player]

                Just token -> do --end game
                              infoDialog window "" $ case token of
                                                          X -> "X won!"
                                                          O -> "O won!"
                                                          None -> "Draw!"
                              close window

        attach widgets trigger y = zipWithM_ (\w x -> set w [on trigger ::= event
(x,y)]) widgets [1..3]

    attach btns command 1
    attach radios select 2
    attach checks command 3

    return window
```

7.5 QtHaskell.hs

```haskell
{-# LANGUAGE RankNTypes, ScopedTypeVariables, EmptyDataDecls #-}
import Control.Monad
import Data.IORef

import Qtc.Classes.Qccs hiding (event)
import Qtc.Classes.Gui hiding (end,move,button)
import Qtc.ClassTypes.Gui
import Qtc.Core.Base
import Qtc.Gui.Base
import Qtc.Gui.QApplication
import Qtc.Gui.QWidget
import Qtc.Gui.QPushButton
import Qtc.Gui.QCheckBox
import Qtc.Gui.QRadioButton
import Qtc.Gui.QDialog
import Qtc.Gui.QGroupBox
import Qtc.Gui.QAbstractButton ()
import Qtc.Gui.QMessageBox
import Qtc.Gui.QLabel
import Qtc.Gui.QVBoxLayout
import Qtc.Gui.QBoxLayout ()
import Qtc.Gui.QGridLayout

import OX

type WidgetCreator = (forall a. QAbstractButton a -> IO ()) -> IO (QWidget ())

-- |Create a button.
data COxQPushButton
type OxPushButton = QPushButtonSc COxQPushButton

oxPushButton :: IO OxPushButton
oxPushButton = qSubClass $ qPushButton " "

button :: WidgetCreator
button e = do
    button <- oxPushButton
    setMaximumSize button (40::Int,40::Int)
    connectSlot button "clicked()" button "click()" e
    qCast_QWidget button

-- |Create a radio button group.
data COxQRadioButton
type OxRadioButton = QRadioButtonSc COxQRadioButton

oxRadioButton :: String -> IO OxRadioButton
oxRadioButton b = qSubClass $ qRadioButton b

radioGroup :: WidgetCreator
radioGroup e = do
    group <- qGroupBox ()

    layout <- qVBoxLayout ()
    setLayout group layout

    button <- oxRadioButton ""
    setChecked button True
    connectSlot button "clicked()" button "click()" $ \button -> setEnabled
(button::OxRadioButton) False

    button_ <- oxRadioButton "?"
    connectSlot button_ "clicked()" button_ "click()" e

    mapM_ (addWidget layout) [button, button_]
    qCast_QWidget group

-- |Create a check box.
data COxQCheckBox
type OxCheckBox = QCheckBoxSc COxQCheckBox

oxCheckBox :: IO OxCheckBox
oxCheckBox = qSubClass $ qCheckBox " "

check :: WidgetCreator
check e = do
    check <- oxCheckBox
    connectSlot check "clicked()" check "click()" e
    qCast_QWidget check

main = do
    --create a new game
    game <- newIORef newGame

    -- create the main window
    qApplication ()
    window <- qDialog ()
    setWindowTitle window "OX"

    vbox <- qVBoxLayout ()
    setLayout window vbox

    grid <- qGridLayout ()
    addLayout vbox grid

    label <- qLabel "Move: X"
    addWidget vbox label

    --attach event handlers
    let event square button = do
            g@(Game player _) <- readIORef game
            setText button $ show player
            let (g'@(Game player _), result) = move g square
            case result of
                Nothing -> do --continue game
                           writeIORef game g'
                           setEnabled button False
                           setText label $ "Move: " ++ show player
                Just token -> do --end game
                              box <- qMessageBox window
                              setText box $ case token of
                                                X -> "X won!"
                                                O -> "O won!"
                                                None -> "Draw!"
                              qshow box ()
        -- widgetCreator will return a widget with an event attached
        -- widgets creates 3 of them in a row
        widgets (widgetCreator::WidgetCreator) y = forM_ [1..3] $ \x -> do
                                       w <- widgetCreator $ event (x,y)
                                       addWidget grid (w, y-1, x-1)

    zipWithM_ widgets [button, radioGroup, check] [1..3]

    -- run the UI
    qshow window ()
    qApplicationExec ()
```

7.6 WxGeneric.hs

```haskell
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses, TemplateHaskell,
UndecidableInstances #-}
import Graphics.UI.WX
import Graphics.UI.WxGeneric
import Graphics.UI.SybWidget.MySYB

data Name = Name {forename :: String, surname :: String} deriving Show
data Student = Student {name :: Name, tutor :: Name, id :: Int} deriving Show

anonymous = Name "" ""
student = Student anonymous anonymous 0

$(derive [''Name,''Student])
instance WxGen Name
instance WxGen Student

main = start $ do
    window <- frame [text := "WxGeneric"]
    editor <- genericWidget window student
    set window [layout := widget editor]
    return window
```

7.7 Banana.hs

```
import Control.Monad
import Data.Maybe
import Graphics.UI.WX hiding (Event)
import Graphics.UI.WXCore hiding (Event)

import Reactive.Banana
import Reactive.Banana.WX

import OX

type State = (Game, Maybe Token)

main = start $ do
    -- create the main window
    window <- frame [text := "OX"]
    label <- staticText window [text := "Move: X"] --overwritten by FRP, here to
ensure correct positioning

    btns <- replicateM 3 $ -> button window [size := sz 40 40]
    radios <- replicateM 3 $ -> radioBox window Vertical ["", "?"] []
    checks <- replicateM 3 $ -> checkBox window [text := "  "] --reserve space for
X/O in label

    set window [layout := column 5 [grid 1 1 [map widget btns, map widget radios,
map widget checks], floatCenter $ widget label]]

    network <- compile $ do
        --convert WxHaskell events to FRP events
        let event0s widgets event = forM widgets $ \x -> event0 x event
        events <- liftM concat $ sequence [event0s btns command, event0s radios
select, event0s checks command]

        let
            moves :: Event (State -> State)
            moves = foldl1 union $ zipWith (\e s -> play s <$ e) events [(x,y) | y
<- [1..3], x <- [1..3]]
                    where play square (game, _) = move game square

            state :: Discrete State
            state = accumD (newGame, Nothing) moves

            player :: Discrete String
            player = (\(Game player _, _) -> show player) <$> state

            tokens :: [Discrete String]
            tokens = map (\e -> stepperD "" (player <@ e)) events

        --wire up the widget event handlers
        zipWithM_ (\b e -> sink b [text :== e, enabled :== null <$> e])
                  (map objectCast btns ++ map objectCast radios ++ map objectCast
checks :: [Control ()])
                  tokens

        sink label [text :== ("Move: " ++) <$> player]

        --end game event handler
        reactimate $ (end window . fromJust) <$> filterE isJust (changes $ snd <$>
state)

    actuate network

end :: Frame () -> Token -> IO ()
end window result = do
    infoDialog window "" $ case result of
                              X -> "X won!"
                              O -> "O won!"
                              None -> "Draw!"
    close window
```

8 Appendix B: Data Binding Source Code

8.1 binding-core

8.1.1 src\Binding\Variable.hs

```
-- | Mutable variables in the IO Monad
module Binding.Variable where

import Data.IORef
import Control.Concurrent.MVar
import Control.Concurrent.STM

class Variable v where
   -- | Create a new variable.
   newVar     :: a -> IO (v a)
   -- | Read a variable.
   readVar    :: v a -> IO a
   -- | Write a variable.
   writeVar   :: v a -> a -> IO ()
   -- | Modify a variable.
   modifyVar  :: v a -> (a -> a) -> IO ()
   -- | Modify a variable, and return some value.
   modifyVar' :: v a -> (a -> (a,b)) -> IO b

instance Variable IORef where
   newVar     = newIORef
   readVar    = readIORef
   writeVar   = writeIORef
   modifyVar  = modifyIORef
   modifyVar' = atomicModifyIORef

instance Variable MVar where
   newVar         = newMVar
   readVar        = takeMVar
   writeVar       = putMVar
   modifyVar v f  = modifyMVar_ v (return . f)
   modifyVar' v f = modifyMVar v (return . f)

instance Variable TVar where
   newVar         = newTVarIO

   readVar        = readTVarIO

   writeVar v x   = atomically $ writeTVar v x

   modifyVar v f  = atomically $ do x <- readTVar v
                                    writeTVar v (f x)

   modifyVar' v f = atomically $ do x <- readTVar v
                                    let (x', y) = f x
                                    writeTVar v x'
                                    return y

instance Variable TMVar where
   newVar         = newTMVarIO

   readVar v      = atomically $ takeTMVar v
   writeVar v x   = atomically $ putTMVar v x

   modifyVar v f  = atomically $ do x <- takeTMVar v
                                    putTMVar v (f x)

   modifyVar' v f = atomically $ do x <- takeTMVar v
                                    let (x', y) = f x
                                    putTMVar v x'
                                    return y
```

8.1.2 src\Binding\Core.hs

```
{-# LANGUAGE ExistentialQuantification #-}
module Binding.Core (module Binding.Variable, Bindable, bind, Source) where

import Binding.Variable

-- | A data binding:
-- @a@ is the type of the data source
-- @a -> d@ is a function that extracts data from the source
-- @t@ is the binding target
-- @d -> t -> IO ()@ is a function that applies data to the target
data Binding a = forall d t. Binding (a -> d) t (t -> d -> IO ())

-- | A simple binding source.
data Source v a = Variable v => Source {bindings :: v [Binding a] -- ^ the
source's bindings
                                       ,var      :: v a}          -- ^ the bound
variable

-- | Update a single binding.
update' :: a -> Binding a -> IO ()
update' source (Binding extract target apply) = apply target $ extract source

-- | Update a binding source's bindings.
update :: Source v a -> IO ()
update (Source bindings source) = do bindings <- readVar bindings
                                     a <- readVar source
                                     mapM_ (update' a) bindings

instance Variable v => Variable (Source v) where
   newVar a = do bindings <- newVar []
                 v <- newVar a
                 return $ Source bindings v

   readVar = readVar . var

   writeVar s a = writeVar (var s) a >> update s

   modifyVar s f = modifyVar (var s) f >> update s

   modifyVar' s f = do b <- modifyVar' (var s) f
                       update s
                       return b

-- | Binding sources.
class Variable b => Bindable b where
```

```
   -- | Create a data binding.
   bind :: b a               -- ^ the binding source
        -> (a -> d)          -- ^ a function that extracts data from the source
        -> t                 -- ^ the binding target
        -> (t -> d -> IO ()) -- ^ a function that applies data to the target
        -> IO ()

instance Variable v => Bindable (Source v) where
   bind (Source bindings var) extract target apply =
      do let binding = Binding extract target apply
         --update the new binding
         a <- readVar var
         update' a binding
         --add the new binding to the list
         modifyVar bindings (binding:)
```

8.1.3 src\Binding\List.hs

```
{-# LANGUAGE ExistentialQuantification #-}
module Binding.List (module Binding.Core, BindingList, toBindingList,
fromBindingList, length, position, seek, seekBy, next, prev, remove', remove,
insert', insert) where

import Prelude hiding (length)
import qualified Prelude as P
import Control.Monad

import Binding.Core

-- | Associates a binding source with a list of data sources.
data BindingList v a = Variable v => BindingList {source :: Source v a -- ^ the
list's binding source
                                                 ,list   :: v [v a]    -- ^ the
bound list
                                                 ,pos    :: v Int}     -- ^ the
current position
-- [v a] is itself in a Variable, to allow for insertions and deletions.

-- | Create a binding list.
toBindingList :: Variable v => [a] -> IO (BindingList v a)
toBindingList [] = error "empty list"
toBindingList list = do list'<- mapM newVar list >>= newVar
                        source <- newVar (head list)
                        pos <- newVar 0
                        return $ BindingList source list' pos

-- | Update the binding list from the 'source'.
update :: BindingList v a -> IO ()
update (BindingList source list pos) = do list' <- readVar list
                                          pos' <- readVar pos
                                          readVar source >>= writeVar (list' !!
pos')

-- | Extract the data from a binding list.
fromBindingList :: Variable v => BindingList v a -> IO [a]
fromBindingList b = do update b
                       readVar (list b) >>= mapM readVar

-- | interface to the binding list's 'Source'
```

```
instance Variable v => Variable (BindingList v) where
   {- WARNING warn "Did you mean to use newBindingList?" -}
   newVar = warn where warn a = toBindingList [a]
   readVar = readVar . source
   writeVar = writeVar . source
   modifyVar = modifyVar . source
   modifyVar' = modifyVar' . source

instance Variable v => Bindable (BindingList v) where
   bind = bind . source

-- | The size of a binding list.
length :: Variable v => BindingList v a -> IO Int
length b = do list <- readVar (list b)
              return $ P.length list

-- | Get the current position.
position :: Variable v => BindingList v a -> IO Int
position b = readVar $ pos b

-- | Bind to a new position in a binding list.
-- Returns the new position; this is convenient for seekBy and friends.
seek:: Variable v => BindingList v a -> Int -> IO Int
seek b new = do pos' <- readVar $ pos b
                if pos' == new then return new else update b >> seek' b new

-- | Unconditional seek. Called after elements have changed position.
seek':: BindingList v a -> Int -> IO Int
seek' (BindingList source list pos) new = do list' <- readVar list
                                             readVar (list' !! new) >>= writeVar
source
                                             writeVar pos new
                                             return new

-- | Bind to a new position in a binding list.
seekBy :: Variable v => (Int -> Int) -> BindingList v a -> IO Int
seekBy f bindingList = do pos <- readVar (pos bindingList)
                          seek bindingList $ f pos

-- | Bind to the next item in a binding list.
next :: Variable v => BindingList v a -> IO Int
next = seekBy succ

-- | Bind to the previous item in a binding list.
prev :: Variable v => BindingList v a -> IO Int
prev = seekBy pred

-- | Remove an element from a list.
remove' :: [a] -> Int ->  [a]
remove' list pos = let (xs, _:ys) = splitAt pos list
                   in xs ++ ys

-- | Remove the current element from the list.
remove :: Variable v => BindingList v a -> IO Int
remove b@(BindingList _ list pos) = do list' <- readVar list
                                       pos' <- readVar pos
                                       writeVar list $ remove' list' pos'
                                       seek' b (if pos' == P.length list' - 1 then
pos' - 1 else pos')
```

```
-- | Insert an element into a list.
insert' :: [a] -> Int -> a -> [a]
insert' list pos x = let (xs, ys) = splitAt pos list
                     in xs ++ [x] ++ ys

-- | Insert an element into the list.
-- The new element is inserted after the current element.
-- This allows appending, but precludes prepending.
insert :: Variable v => BindingList v a -> a -> IO Int
insert b@(BindingList _ list pos) x = do update b
                                         list' <- readVar list
                                         pos' <- readVar pos
                                         x' <- newVar x
                                         let pos'' = pos' + 1
                                         writeVar list $ insert' list' pos'' x'
                                         seek' b pos''
```

8.1.4 tests\HUnit.hs

```
{-# LANGUAGE TupleSections #-}
import Test.HUnit

import Control.Monad
import Data.IORef
import System.Exit
import System.Random

import Binding.List as B
import Prelude as P

-- Change these to exercise different variable and data types
type V = IORef
type A = Int

-- *** Test pure helpers ***

-- | Generate a list for testing.
-- Many operations are expected to fail on lists of less than 2 elements.
list' :: IO ([A], Int)
list' = do size <- randomRIO (2,100)
           list <- replicateM size randomIO
           return (list, size)

testRemove' :: Assertion
testRemove' = do (list, size) <- list'
                 pos <- randomRIO (0, size-2)
                 let actual = remove' list pos
                 assertEqual "List hasn't shrunk correctly" (size-1) (P.length
actual)
                 assertEqual "Head of list incorrect" (take pos list) (take pos
actual)
                 assertEqual "Tail of list incorrect" (drop (pos+1) list) (drop
pos actual)

testRemoveLast' :: Assertion
testRemoveLast' = do (list, size) <- list'
                     let actual = remove' list (size-1)
                     assertEqual "List hasn't shrunk correctly" (size-1) (P.length
actual)
                     assertEqual "List is incorrect" (take (size-1) list) actual

testInsert' :: Assertion
testInsert' = do (list, size) <- list'
                 pos <- randomRIO (0, size-1)
                 new <- randomIO
                 let actual = insert' list pos new
                 assertEqual "List hasn't shrunk correctly" (size+1) (P.length
actual)
                 assertEqual "Head of list incorrect" (take pos list) (take pos
actual)
                 assertEqual "Element not inserted" new (actual !! pos)
                 assertEqual "Tail of list incorrect" (drop pos list) (drop
(pos+1) actual)

--- *** Test monadic functions ***
```

```haskell
testSource :: Assertion
testSource = do --bind a source
                expected <- randomIO
                source <- newVar expected :: IO (Source V A)
                target <- randomIO >>= newVar :: IO (Source V A)
                bind source id target writeVar
                actual <- readVar target
                assertEqual "Initial Bind" expected actual
                --change its value
                expected <- randomIO
                writeVar source expected
                actual <- readVar target
                assertEqual "Value Changed" expected actual

-- | Generate a 'BindingList' for testing.
list :: IO ([A], Int, BindingList V A)
list = do (list, size) <- list'
          liftM (list, size,) (toBindingList list)

-- | Assert that a 'BindingList' holds the expected list.
assertList :: [A] -> BindingList V A -> Assertion
assertList list bl = fromBindingList bl >>= (list @=?)

-- | Assert that a 'BindingList' holds the expected list.
assertPos :: Int -> BindingList V A -> Int -> Assertion
assertPos expected bl reported = do pos <- position bl
                                    assertEqual "Wrong positon" expected pos
                                    assertEqual "Wrong positon reported" pos
reported

testList :: Assertion
testList = do (expected, _, bl) <- list
              assertList expected bl

testLength :: Assertion
testLength = do (_, expected, bl) <- list
                B.length bl >>= (expected @=?)

testSeek :: Assertion
testSeek = do (list, size, bl) <- list
              pos <- randomRIO (0,size-1)
              seek bl pos >>= assertPos pos bl
              actual <- readVar bl
              list !! pos @=? actual

testSeekBy :: Assertion
testSeekBy = do (_, size, bl) <- list
                init <- randomRIO (0, size-1)
                offset <- randomRIO (-init, size-init-1)
                let expected = init + offset
                seek bl init
                actual <- seekBy (offset+) bl
                --give a more detailed error message than assertPos
                assertEqual ("Seek from " ++ show init ++ " by " ++ show offset)
expected actual
                assertPos expected bl actual

testNext :: Assertion
testNext = do (_, size, bl) <- list
              init <- randomRIO (0, size-2)
              seek bl init
              B.next bl >>= assertPos (init+1) bl

testPrev :: Assertion
testPrev = do (_, size, bl) <- list
              init <- randomRIO (1, size-1)
              seek bl init
              prev bl >>= assertPos (init-1) bl

testRemove :: Assertion
testRemove = do (list, size, bl) <- list
                pos <- randomRIO (0, size-2)
                seek bl pos
                remove bl >>= assertPos pos bl
                assertList (remove' list pos) bl

testRemoveLast :: Assertion
testRemoveLast = do (list, size, bl) <- list
                    seek bl (size-1)
                    remove bl >>= assertPos (size-2) bl
                    assertList (remove' list (size-1)) bl

testInsert :: Assertion
testInsert = do (list, size, bl) <- list
                pos <- randomRIO (0, size-1)
                new <- randomIO
                seek bl pos
                let pos' = pos+1
                insert bl new >>= assertPos pos' bl
                assertList (insert' list pos' new) bl

main = do Counts _ _ e f <- runTestTT $ TestList
             ["Source" ~: testSource
             ,"binding lists" ~: testList
             ,"length" ~: testLength
             ,"seek" ~: testSeek
             ,"seekBy" ~: testSeekBy
             ,"next" ~: testNext
             ,"prev" ~: testPrev
             ,"remove'" ~: testRemove'
             ,"remove" ~: testRemove
             ,"remove' last" ~: testRemoveLast'
             ,"remove last" ~: testRemoveLast
             ,"insert'" ~: testInsert'
             ,"insert" ~: testInsert]
          when (e>0 || f>0) exitFailure
```

8.1.5 tests\QuickCheck.hs

```
{-# LANGUAGE TupleSections, TemplateHaskell #-}
import Test.QuickCheck
import Test.QuickCheck.Modifiers
import Test.QuickCheck.Monadic
import Test.QuickCheck.All
import Test.QuickCheck.Test

import Control.Monad
import Data.IORef
import System.Exit

import Binding.List as B
import Prelude as P

-- Change these to exercise different variable and data types
type V = IORef
type A = Char

-- *** Helpers to generate random lists and positions ***

-- | A random list with at least two elements.
newtype List = List [A] deriving Show

instance Arbitrary List where
   arbitrary = liftM List $ choose (2, 100) >>= vector
   shrink (List xs) = [List ys | ys <- shrink xs, P.length ys > 1]

-- | Maps @i@ to a position in @xs@.
anywhere :: Int -> [A] -> Int
anywhere i xs = let max = P.length xs - 1
                in if max == 0 then 0 else i `mod` max

-- | Anywhere in the list except the last element.
notLast :: Int -> [A] -> Int
notLast i = anywhere i . tail

-- | Create a 'BindingList', and 'seek' to @pos@.
list :: [A] -> Int -> IO (BindingList V A)
list xs pos = do bl <- toBindingList xs
                 seek bl pos
                 return bl

-- *** Test pure functions ***

prop_remove' :: [A] -> Int -> Bool
prop_remove' xs i = let pos = anywhere i xs
                        actual = remove' xs pos
                    in P.length actual == P.length xs - 1
                    && take pos actual == take pos xs
                    && drop (pos+1) xs == drop pos actual

prop_removeLast' :: [A] -> Bool
prop_removeLast' xs = let pos = P.length xs - 1
                          actual = remove' xs pos
                      in P.length actual == pos
                      && actual == take pos xs
```

```haskell
prop_insert' :: [A] -> Int -> A -> Bool
prop_insert' xs i x = let pos = anywhere i xs
                          actual = insert' xs pos x
                      in P.length actual == P.length xs + 1
                      && take pos actual == take pos xs
                      && actual !! pos == x
                      && drop pos actual == drop (pos+1) xs

-- *** QuickCheck 'Property's for Monadic actions. ***

prop_Source :: (A,A,A) -> Property
prop_Source (a,b,c) = monadicIO $ do
   (x,y) <- run $ do --bind a source
                  source <- newVar a :: IO (Source V A)
                  target <- newVar c :: IO (Source V A)
                  bind source id target writeVar
                  x <- readVar target
                  --change its value
                  writeVar source b
                  y <- readVar target
                  return (x,y)
   assert (x==a && y==b)

prop_List :: NonEmptyList A -> Property
prop_List (NonEmpty xs) = monadicIO $ do
   ys <- run $ (toBindingList xs :: IO (BindingList V A)) >>= fromBindingList
   assert (ys == xs)

prop_length :: NonEmptyList A -> Property
prop_length (NonEmpty xs) = monadicIO $ do
   l <- run $ (toBindingList xs :: IO (BindingList V A)) >>= B.length
   assert (l == P.length xs)

prop_seek :: NonEmptyList A -> Int -> Property
prop_seek (NonEmpty xs) i = let pos = anywhere i xs in monadicIO $ do
   (new, x) <- run $ do bl <- toBindingList xs :: IO (BindingList V A)
                        liftM2 (,) (seek bl pos) (readVar bl)
   assert (new == pos && x == xs !! pos)

prop_position :: NonEmptyList A -> Int -> Property
prop_position (NonEmpty xs) i = let pos = anywhere i xs in monadicIO $ do
   new <- run $ list xs pos >>= position
   assert (new == pos)

prop_seekBy :: List -> Int -> Int -> Property
prop_seekBy (List xs) a b = let size = P.length xs
                                init = anywhere a xs
                                offset = anywhere b xs - init
                            in monadicIO $ do
   (new, x) <- run $ do bl <- list xs init
                        liftM2 (,) (seekBy (offset+) bl) (readVar bl)
   assert (new == init + offset && x == xs !! new)

prop_next :: List -> Int -> Property
prop_next (List xs) i = let pos = notLast i xs in monadicIO $ do
   (new, x) <- run $ do bl <- list xs pos
                        liftM2 (,) (B.next bl) (readVar bl)
   assert (new == pos + 1 && x == xs !! new)
```

```haskell
prop_prev :: List -> Int -> Property
prop_prev (List xs) i = let pos = anywhere i xs + 1 in monadicIO $ do
   (new, x) <- run $ do bl <- list xs pos
                        liftM2 (,) (prev bl) (readVar bl)
   assert (new == pos - 1 && x == xs !! new)

prop_insert :: List -> Int -> A -> Property
prop_insert (List xs) i x = let pos = anywhere i xs
                                new = pos + 1
                            in monadicIO $ do
   (pos', ys) <- run $ do bl <- list xs pos
                          liftM2 (,) (insert bl x) (fromBindingList bl)
   assert (ys == insert' xs new x && pos' == new)

-- we test removing the last element separately because it's a special case
testRemove :: [A] -> Int -> PropertyM IO (Int, [A])
testRemove xs pos = run $ do bl <- list xs pos
                             liftM2 (,) (remove bl) (fromBindingList bl)

prop_remove :: List -> Int -> Property
prop_remove (List xs) i = let pos = notLast i xs in monadicIO $ do
   (pos', ys) <- testRemove xs pos
   assert (ys == remove' xs pos && pos' == pos)

prop_removeLast :: List -> Property
prop_removeLast (List xs) = let pos = P.length xs - 1 in monadicIO $ do
   (pos', ys) <- testRemove xs pos
   assert (ys == remove' xs pos && pos' == pos -1)

-- | Test the 'Property's
main = do passed <- $quickCheckAll
          unless passed exitFailure
```

8.2 binding-gtk

8.2.1 src\Binding\Gtk.hs

```haskell
{-# LANGUAGE FlexibleContexts #-}
module Binding.Gtk where

import Control.Monad
import Control.Monad.Trans
import Graphics.UI.Gtk

import Binding.List as B

-- | Bind a 'Source' to a control.
bindToControl :: Bindable b =>
                 b a      -- ^ the binding source
              -> (a -> d) -- ^ a function that extracts data from the source
              -> c        -- ^ the target control
              -> Attr c d -- ^ the attribute of the control to bind to
              -> IO ()
bindToControl source extract control attribute = bind source extract control (\c d
-> set c [attribute := d])

-- | Bind from a control to a 'Source'.
-- The source is updated when the control loses focus.
bindFromControl :: (WidgetClass c, Bindable b) =>
                   c             -- ^ the control
                -> Attr c d      -- ^ the attribute of the control to bind from
                -> (a -> d -> a) -- ^ a function that applies data from the
control to the source
                -> b a           -- ^ the binding source
                -> IO (ConnectId c)
bindFromControl control attribute apply source =
   control `on` focusOutEvent $ liftIO $ do d <- get control attribute
                                            a <- readVar source
                                            writeVar source (apply a d)
                                            return False

-- | Create a two-way data binding.
bindControl :: (WidgetClass c, Bindable b) =>
               b a           -- ^ the binding source
            -> (a -> d)      -- ^ a function that extracts data from the source
            -> c             -- ^ the control
            -> Attr c d      -- ^ the attribute of the control to bind to
            -> (a -> d -> a) -- ^ a function that applies data from the control to
the source
            -> IO (ConnectId c)
bindControl source extract control attribute apply = do
   bindToControl source extract control attribute
   bindFromControl control attribute apply source

-- | Create a simple two-way data binding for a 'Textual' control.
bindTextEntry :: (Show a, Read a, EntryClass c, WidgetClass c, Bindable b) =>
                 b a -- ^ the binding source
              -> c   -- ^ the control
              -> IO (ConnectId c)
bindTextEntry source control = do
   bindToControl source show control entryText
   control `on` focusOutEvent $ liftIO $ do d <- get control entryText
                                            writeVar source (read d)
                                            return False

-- | Create a set of navigation buttons for a binding list.
navigation :: Variable v =>
              BindingList v a -- ^ the binding list
           -> a               -- ^ the default value for inserts
           -> IO HButtonBox
navigation bl new = do spin <- spinButtonNewWithRange 0 1 1
                       let setRange = B.length bl >>= spinButtonSetRange spin 0 .
fromIntegral . pred
                       setRange
                       afterValueSpinned spin $ spinButtonGetValueAsInt spin >>=
seek bl >> return ()
                       buttons <- forM [("<<", spinButtonSetValue spin 0)
                                       ,(">>", spinButtonSpin spin SpinEnd 0)
                                       ,("+", insert bl new >>= spinButtonSetValue
spin . fromIntegral >> setRange)
                                       ,("-", B.remove bl >>= spinButtonSetValue
spin . fromIntegral >> setRange)]
                                       $ \(l,c) -> do b <- buttonNewWithLabel l
                                                      on b buttonActivated c
                                                      return b

                       let del = last buttons
                       del `on` buttonActivated $ do l <- B.length bl
                                                     del `set` [widgetSensitive :=
l > 1]

                       (buttons !! 2) `on` buttonActivated $ del `set`
[widgetSensitive := True] --"+"

                       box <- hButtonBoxNew
                       containerAdd box spin
                       mapM_ (containerAdd box) buttons
                       return box
```

8.2.2 demo\simple.hs

```
import Data.IORef
import Graphics.UI.Gtk

import Binding.Core
import Binding.Gtk

main = do --create widgits
          initGUI
          text1 <- entryNew
          text2 <- entryNew
          --bind them
          source <- newVar 0 :: IO (Source IORef Double)
          bindTextEntry source text1
          bindTextEntry source text2
          --arrange the widgits
          hBox <- hBoxNew True 0
          boxPackStartDefaults hBox text1
          boxPackStartDefaults hBox text2
          --create the main window
          window <- windowNew
          set window [containerChild := hBox, windowTitle := "Data Binding with
Gtk2Hs"]
          onDestroy window mainQuit
          --start the application
          widgetShowAll window
          mainGUI
```

8.2.3 demo\lists.hs

```
import Data.IORef
import Control.Monad
import Graphics.UI.Gtk

import Binding.List
import Binding.Gtk

data Person = Person {name::String, age::Int, active::Bool} deriving (Read, Show)

main = do -- read the input
          f <- readFile "in.txt"
          bl <- toBindingList $ read f :: IO (BindingList IORef Person)
          --create widgits
          initGUI
          name' <- entryNew
          age' <- spinButtonNewWithRange 0 120 1
          active' <- checkButtonNew
          --bind them
          nav <- navigation bl $ Person "" 0 False
          bindControl bl name name' entryText (\p n -> p {name = n})
          bindControl bl (fromIntegral . age) age' spinButtonValue (\p a -> p {age
= round a})
          bindControl bl active active' toggleButtonActive (\p a -> p {active =
a})
          --arrange the widgits
          table <- tableNew 3 2 True

          zipWithM_ (\cap row -> do label <- labelNew $ Just cap
                                    tableAttachDefaults table label 0 1 row
(row+1))
                    ["Name:", "Age:", "Active:"] [0..2]

          zipWithM_ (\wid row -> tableAttachDefaults table wid 1 2 row (row+1))
                    [toWidget name', toWidget age', toWidget active'] [0..2]

          vBox <- vBoxNew False 0
          boxPackStartDefaults vBox table
          boxPackStartDefaults vBox nav
          -- create the main window
          window <- windowNew
          set window [containerChild := vBox, windowTitle := "Data Binding with
Gtk2Hs"]
          onDestroy window mainQuit
          --start the application
          widgetShowAll window
          mainGUI
          new <- fromBindingList bl
          writeFile "out.txt" $ show new
```

8.3 binding-wx

8.3.1 src\Binding\Wx.hs

```haskell
{-# LANGUAGE RankNTypes #-}
module Binding.Wx where

import Control.Monad
import Graphics.UI.WX

import Binding.List as B

-- | Bind a 'Source' to a control.
bindToControl :: Bindable b =>
                 b a      -- ^ the binding source
              -> (a -> d) -- ^ a function that extracts data from the source
              -> c        -- ^ the target control
              -> Attr c d -- ^ the attribute of the control to bind to
              -> IO ()
bindToControl source extract control attribute = bind source extract control (\c d
-> set c [attribute := d])

-- | Bind from a control to a 'Source'.
-- The source is updated when the control loses focus.
bindFromControl :: (Bindable b, Reactive c) =>
                   c             -- ^ the control
                -> Attr c d      -- ^ the attribute of the control to bind from
                -> (a -> d -> a) -- ^ a function that applies data from the
control to the source
                -> b a           -- ^ the binding source
                -> IO ()
bindFromControl control attribute apply source =
   set control [on focus := \f -> unless f $ do d <- get control attribute
                                                a <- readVar source
                                                writeVar source (apply a d)
                                                propagateEvent]

-- | Create a two-way data binding.
bindControl :: (Bindable b, Reactive c) =>
               b a           -- ^ the binding source
            -> (a -> d)      -- ^ a function that extracts data from the source
            -> c             -- ^ the control
            -> Attr c d      -- ^ the attribute of the control to bind to
            -> (a -> d -> a) -- ^ a function that applies data from the control to
the source
            -> IO ()
bindControl source extract control attribute apply = do
   bindToControl source extract control attribute
   bindFromControl control attribute apply source

-- | Create a simple two-way data binding for a 'Textual' control.
bindTextual :: (Show a, Read a, Bindable b, Textual c, Reactive c) =>
               b a -- ^ the binding source
            -> c   -- ^ the control
            -> IO ()
bindTextual source control = do
   bindToControl source show control text
   set control [on focus := \f -> unless f $ do d <- get control text
```

```
                                                          writeVar source (read d)
                                                          propagateEvent]

-- | Create a set of navigation buttons for a binding list.
navigation owner bl new = do spin <- spinCtrl owner 0 1 [on select ::= \s -> get s
selection >>= seek bl >> return ()]
                             let setRange = B.length bl >>= spinCtrlSetRange spin
0 . pred
                             setRange
                             let go i = spin `set` [selection := i] >> seek bl i
                             buttons <- forM [("<<", go 0 >> return ())
                                             ,(">>", B.length bl >>= go . pred >>
return ())
                                             ,("+", insert bl new >>= go >>
setRange)
                                             ,("-", remove bl >>= go >> setRange)]
                                             $ \(t,c) -> button owner [text := t,
on command := c]

                             let del = last buttons
                             del `set` [on command :~ (>> do l <- B.length bl
                                                             del `set` [enabled :=
l > 1])                                                               ]

                             (buttons !! 2) `set` [on command :~ (>> del `set`
[enabled := True])] --"+"

                             return $ row 0 $ widget spin : map widget buttons
```

8.3.2 demo\simple.hs

```
import Data.IORef
import Graphics.UI.WX

import Binding.Core
import Binding.Wx

main = start $ do --create widgits
                  window <- frame [text := "Data Binding with Gtk2Hs"]
                  text1 <- entry window []
                  text2 <- entry window []
                  --bind them
                  source <- newVar 0 :: IO (Source IORef Double)
                  bindTextual source text1
                  bindTextual source text2
                  --start the application
                  set window [layout := row 0 [widget text1, widget text2]]
```

8.3.3 demo\lists.hs

```
import Control.Monad
import Data.IORef
import Data.List
import Graphics.UI.WX

import Binding.List
import Binding.Wx

data Person = Person {name::String, age::Int, active::Bool} deriving (Read, Show)

main = do -- read the input
          f <- readFile "in.txt"
          bl <- toBindingList $ read f :: IO (BindingList IORef Person)
          start $ do --create widgits
                     window <- frame [text := "Data Binding with WxHaskell"]
                     name' <- entry window []
                     age' <- spinCtrl window 0 120 []
                     active' <- checkBox window []
                     --bind them
                     nav <- navigation window bl $ Person "" 0 False
                     bindControl bl name name' text (\p n -> p {name = n})
                     bindControl bl (fromIntegral . age) age' selection (\p a -> p
{age = a})
                     bindControl bl active active' checked (\p a -> p {active =
a})
                     --arrange the widgits
                     let labels = map (floatRight . label) ["Name:", "Age:",
"Active:"]
                     let widgets = map floatLeft [widget name', widget age',
widget active']
                     --start the application
                     set window [layout := column 10 [grid 10 10 $ transpose
[labels, widgets], nav]
                                ,on closing := fromBindingList bl >>= \l ->
writeFile "out.txt" (show l) >> propagateEvent]
```

9 Appendix C: Data Binding Documentation

Running Haddock [**14**] over the source in Appendix B: Data Binding Source Code will produce nicely formatted HTML documentation.

The text of Haddock’s documentation is reproduced here for reference.

9.1 Binding.Variable

```
class Variable v where

      Methods

      newVar :: a -> IO (v a)

          Create a new variable.

      readVar :: v a -> IO a

          Read a variable.

      writeVar :: v a -> a -> IO ()

          Write a variable.

      modifyVar :: v a -> (a -> a) -> IO ()

          Modify a variable.

      modifyVar' :: v a -> (a -> (a, b)) -> IO b

          Modify a variable, and return some value.

      Instances
             Variable TVar
             Variable IORef
             Variable MVar
             Variable TMVar
             Variable v => Variable (Source v)
             Variable v => Variable (BindingList v)  interface to the binding
             list's Source
```

9.2 Binding.Core

```
class Variable b => Bindable b where

       Binding sources.

       Methods

       bind
           :: b a the binding source
           -> (a -> d)            a function that extracts data from the source
           -> t                   the binding target
           -> (t -> d -> IO ())   a function that applies data to the target
           -> IO ()

           Create a data binding.

       Instances
           Variable v => Bindable (Source v)
           Variable v => Bindable (BindingList v)

data Source v a

       A simple binding source.

       Instances
           Variable v => Variable (Source v)
           Variable v => Bindable (Source v)
```

9.3 Binding.List

```
data BindingList v a

       Associates a binding source with a list of data sources.

       Instances
           Variable v => Variable (BindingList v) interface to the binding list's
           Source
           Variable v => Bindable (BindingList v)

toBindingList :: Variable v => [a] -> IO (BindingList v a)

       Create a binding list.

fromBindingList :: Variable v => BindingList v a -> IO [a]

       Extract the data from a binding list.

length :: Variable v => BindingList v a -> IO Int

       The size of a binding list.

position :: Variable v => BindingList v a -> IO Int

       Get the current position.

seek :: Variable v => BindingList v a -> Int -> IO Int

       Bind to a new position in a binding list. Returns the new position; this is
       convenient for seekBy and friends.

seekBy :: Variable v => (Int -> Int) -> BindingList v a -> IO Int

       Bind to a new position in a binding list.

next :: Variable v => BindingList v a -> IO Int

       Bind to the next item in a binding list.

prev :: Variable v => BindingList v a -> IO Int

       Bind to the previous item in a binding list.

remove' :: [a] -> Int -> [a]

       Remove an element from a list.

remove :: Variable v => BindingList v a -> IO Int

       Remove the current element from the list.

insert' :: [a] -> Int -> a -> [a]

       Insert an element into a list.
```

```
insert :: Variable v => BindingList v a -> a -> IO Int
```

    Insert an element into the list. The new element is inserted after the
    current element. This allows appending, but precludes prepending.

9.4 Binding.Gtk

```
bindToControl
       :: Bindable b
       => b a        the binding source
       -> (a -> d)   a function that extracts data from the source
       -> c          the target control
       -> Attr c d   the attribute of the control to bind to
       -> IO ()

       Bind a Source to a control.

bindFromControl
       :: (WidgetClass c, Bindable b)
       => c                  the control
       -> Attr c d           the attribute of the control to bind from
       -> (a -> d -> a)      a function that applies data from the control to the
       source
       -> b a                the binding source
       -> IO (ConnectId c)

       Bind from a control to a Source. The source is updated when the control
       loses focus.

bindControl
       :: (WidgetClass c, Bindable b)
       => b a                the binding source
       -> (a -> d)           a function that extracts data from the source
       -> c                  the control
       -> Attr c d           the attribute of the control to bind to
       -> (a -> d -> a)      a function that applies data from the control to the
       source
       -> IO (ConnectId c)

       Create a two-way data binding.

bindTextEntry
       :: (Show a, Read a, EntryClass c, WidgetClass c, Bindable b)
       => b a        the binding source
       -> c          the control
       -> IO (ConnectId c)

       Create a simple two-way data binding for a Textual control.

navigation
       :: Variable v
       => BindingList v a  the binding list
       -> a                the default value for inserts
       -> IO HButtonBox

       Create a set of navigation buttons for a binding list.
```

9.5 Binding.Wx

```
bindToControl
       :: Bindable b
       => b a        the binding source
       -> (a -> d)   a function that extracts data from the source
       -> c          the target control
       -> Attr c d   the attribute of the control to bind to
       -> IO ()

       Bind a Source to a control.

bindFromControl
       :: (Bindable b, Reactive c)
       => c                the control
       -> Attr c d         the attribute of the control to bind from
       -> (a -> d -> a)    a function that applies data from the control to the
       source
       -> b a              the binding source
       -> IO ()

       Bind from a control to a Source. The source is updated when the control
       loses focus.

bindControl
       :: (Bindable b, Reactive c)
       => b a              the binding source
       -> (a -> d)         a function that extracts data from the source
       -> c                the control
       -> Attr c d         the attribute of the control to bind to
       -> (a -> d -> a)    a function that applies data from the control to the
       source
       -> IO ()

       Create a two-way data binding.

bindTextual
       :: (Show a, Read a, Bindable b, Textual c, Reactive c)
       => b a        the binding source
       -> c          the control
       -> IO ()

       Create a simple two-way data binding for a Textual control.

navigation
       :: Variable v
       => Window w         the buttons' owner
       -> BindingList v a  the binding list
       -> a                the default value for inserts
       -> IO Layout

Create a set of navigation buttons for a binding list. WxHaskell cannot change a
spin control's range after it has been created, hence the maximum value will be
incorrect following an insert or delete.
```

10 Bibliography

[1] Simon Peyton Jones, "Tackling the Awkward Squad," in *Engineering theories of software construction*. Marktoberdorf: IOS Press, 2001.

[2] Brad A. Myers, "Why are human-computer interfaces difficult to design and implement?," Computer Science Department, Carnegie-Mellon University, Pittsburgh, CMU-CS-93-183, July 1993.

[3] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler, "A History of Haskell: Being LazyWith Class," , San Diego, 2007.

[4] Simon Marlow. Haskell 2010 Language Report. [Online]. http://haskell.org/haskellwiki/Language_and_library_specification

[5] Konstantin Läufer, "Type Classes with Existential Types," *Journal of Functional Programming*, vol. 6, no. 3, pp. 485-518, 1996.

[6] Conor Mcbride and Ross Paterson, "Applicative Programming with Effects," *Journal of Functional Programming*, vol. 18, no. 1, pp. 1-13, January 2008.

[7] John Hughes, "Generalising Monads to Arrows ," *Science of Computer Programming*, vol. 37, no. 1-3, pp. 67-111, May 2000.

[8] Ross Paterson, "Arrows and Computation," in *The Fun of Programming*, Jeremy Gibbons and Oege de Moor, Eds. Basingstoke, UK: Palgrave Macmillan, 2003, ch. 10.

[9] Mark P. Jones. Hugs 98. [Online]. http://www.haskell.org/hugs/

[10] York Haskell Compiler. [Online]. http://www.haskell.org/haskellwiki/Yhc

[11] Utrecht University, Department of Information and Computing Sciences. Utrecht Haskell Compiler. [Online]. http://www.cs.uu.nl/wiki/UHC

[12] The Glasgow Haskell Compiler. [Online]. http://www.haskell.org/ghc/

[13] Hackage. [Online]. http://hackage.haskell.org/

[14] Haddock. [Online]. http://www.haskell.org/haddock/

[15] The Haskell Platform. [Online]. http://hackage.haskell.org/platform/

[16] GTK+. [Online]. http://www.gtk.org/

[17] Glade. [Online]. http://glade.gnome.org/

[18] Gtk2Hs. [Online]. http://www.haskell.org/gtk2hs/

[19] wxWidgets. [Online]. http://www.wxwidgets.org/

[20] Julian Smart, Kevin Hock, and Stefan Csomor, *Cross-platform GUI programming with wxWidgets*. United States of America: Prentice Hall, 2006.

[21] Daan Leijen, "wxHaskell: a portable and concise GUI library for haskell," in *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Snowbird, 2004, pp. 57-68.

[22] Thomas van Noort. Building GUIs in Haskell. [Online]. http://www.cs.ru.nl/~thomas/publications/noot07-building-guis-in.pdf

[23] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy, "Composable Memory Transactions," in *ACM Conference on Principles and Practice of Parallel Programming*, Chicago, June 2005.

[24] Syed K. Shahzad, Michael Granitzer, and Klause Tochterman, "Designing User Interfaces through Ontological User Model: Functional Programming Approach," in *Fourth International Conference on Computer Sciences and Convergence Information Technology*, Seoul, 2009, pp. 99-104.

[25] Qt. [Online]. http://qt.nokia.com/

[26] Matthias Kalle Dalheimer, *Programming with Qt*, 2nd ed., Ariane Hesse, Ed. United States of America: O'Reilley, 2002.

[27] Johan Thelin, *Foundations of Qt development*, Jason Gilmore, Ed. New York, United States of America: Apress, 2007.

[28] David Harley. qtHaskell. [Online]. http://qthaskell.berlios.de/

[29] Wolfgang Jeltsch. (2008, October) Qt-style C++ in Haskell. [Online]. http://softbase.org/hqk/qoo/qoo.pdf

[30] Wolfgang Jeltsch. HQK. [Online]. http://www.haskell.org/haskellwiki/HQK

[31] Magnus Carlsson and Thomas Hallgren, "FUDGETS: a graphical user interface in a lazy functional language," in *Proceedings of the conference on Functional programming languages and computer architecture*, Copenhagen, 1993, pp. 321-330.

[32] Conal Elliott and Paul Hudak, "Functional Reactive Animation," in *Proceedings of the Second ACM SIGPLAN International Conference on Functional programming*, vol. 32, Amsterdam, August 1997, pp. 263-273.

[33] Zhanyong Wan and Paul Hudak, "Functional Reactive Programming from First Principles," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, 2000, pp. 242-252.

[34] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson, "Arrows, Robots, and Functional Reactive Programming," *Lecture Notes in Computer Science*, vol. 2638, pp. 159-187, August 2002.

[35] Henrik Nilsson, Antony Courtney, and John Peterson, "Functional Reactive Programming, Continued," in *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, Pittsburgh, 2002, pp. 51-64.

[36] Antony Courtney, Henrik Nilsson, and John Peterson, "The Yampa Arcade," in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, New York, 2003, pp. 7-18.

[37] Antony Courtney and Conal Elliott, "Genuinely Functional User Interfaces," in *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Firenze, September 2001, pp. 41-69.

[38] Antony Courtney, "Functionally Modeled User Interfaces," in *Interactive systems: design, specification, and verification: 10th international workshop*, Funchal, 2003, pp. 107-123.

[39] Bart Robinson. (2004, May) WxFruit: A Practical GUI Toolkit for Functional Reactive Programming. [Online]. http://web.archive.org/web/20071224143245/http://zoo.cs.yale.edu/classes/cs490/03-04b/bartholomew.robinson/wxfruit.pdf

[40] John Peterson, Antony Courtney, and Bart Robinson, "Can GUI Programming Be Liberated From The IO Monad," in *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Snowbird, 2004.

[41] Remi Turk. (2007, March) Juicy Fruit. [Online]. http://www.cs.uu.nl/wiki/pub/Afp0607/DomainSpecificLanguages/fruit.pdf

[42] Conal Elliott, "Push-Pull Functional Reactive Programming," in *Proceedings of the Second ACM SIGPLAN Symposium on Haskell*, Edinburgh, 2009, pp. 25-36.

[43] F. Warren Burton, "Indeterminate behavior with determinate semantics in parallel programs," in *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, New York, 1989, pp. 340-346.

[44] F. Warren Burton, "Encapsulating non-determinacy in an abstract data type with determinate semantics," *Journal of Functional Programming*, vol. 1, no. 1, pp. 3-20, 1991.

[45] Conal Elliot. (2007) Applicative Data-Driven Computation. [Online]. http://conal.net/papers/data-driven/paper.pdf

[46] Wolfgang Jeltsch, "Improving Push-based FRP," in *Ninth Symposium on Trends in Functional Programming*, America, Netherlands, 2008, p. Chapter 14.

[47] Wolfgang Jeltsch, "Signals, Not Generators!," in *Tenth Symposium on Trends in Functional Programming*, Komárno, 2009, p. Chapter 22.

[48] Wolfgang Jeltsch. (2008, May) Declarative Programming Of Interactive Systems With Grapefruit. [Online]. http://www.informatik.tu-cottbus.de/~jeltsch/research/uustc-20080529-slides.pdf

[49] Heinrich Apfelmus. Reactive Banana. [Online]. http://www.haskell.org/haskellwiki/Reactive-banana

[50] Mike Potel. (1996) MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java. [Online]. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

[51] Trygve M. H. Reenskaug. (1979) MVC. [Online]. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[52] Leksah. [Online]. http://leksah.org/

[53] JP Moresmau, Thomas ten Cate, and Leif Frenzel. EclipseFP. [Online]. http://eclipsefp.sourceforge.net/

[54] Thomas ten Cate. (2009, August) EclipseFP GSoC. [Online]. http://eclipsefp.wordpress.com/2009/08/25/endgame/

[55] Johan Tibell. (2010, August) Results from the State of Haskell, 2010 Survey. [Online]. http://blog.johantibell.com/2010/08/results-from-state-of-haskell-2010.html

[56] Gtk2Hs Trac. [Online]. http://hackage.haskell.org/trac/gtk2hs/ticket/1203

[57] Dean Herington. HUnit. [Online]. http://hunit.sourceforge.net/

[58] Koen Claessen and John Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *SIGPLAN*, vol. 35, no. 9, pp. 268-279, September 2000.

[59] Koen Claessen and John Hughes, "Testing monadic code with QuickCheck," *SIGPLAN*, vol. 37, no. 12, pp. 47-59, December 2002.

[60] GHC Library Reference. IOArray. [Online]. http://www.haskell.org/ghc/docs/latest/html/libraries/array/Data-Array-IO.html

[61] Donald Knuth, "Structured Programming with go to Statements," *ACM Computing Surveys*, vol. 6, no. 4, pp. 261-301, December 1974.

[62] Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.

[63] David A. Patterson and John L. Hennessy, "The Difficulty of Creating Parallel Processing Programs," in *Computer Organization and Design: The Hardware/Software Interface*.: Morgan Kaufmann, 2008, ch. 7.2, pp. 634-638.

[64] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly, "System F with Type Equality Coercions," in *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, Nice, 2007, pp. 53-66.

[65] Microsoft. Data Binding and Windows Forms. [Online]. http://msdn.microsoft.com/en-us/library/c8aebh9k.aspx

[66] Ted Neward. (2006, June) The Vietnam of Computer Science. [Online]. http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx