

# Building GUIs in Haskell\*

## Comparing Gtk2Hs and wxHaskell

Thomas van Noort  
Center for Software Technology  
Universiteit Utrecht, The Netherlands  
trnoort@cs.uu.nl

### ABSTRACT

*This paper discusses which features of Haskell make the language suitable for building GUIs. Two monadic libraries, Gtk2Hs and wxHaskell, are presented to illustrate that it is not hard to build a GUI in Haskell. A running example is used to compare the two libraries, and their usability in particular. Furthermore, an extension for both libraries is discussed to further improve the usability: the visual GUI builder Glade for Gtk2Hs, and the eXtended and Typed Control (XTC) library for wxHaskell.*

### 1. INTRODUCTION

Building GUIs in Haskell is believed to be hard. Partly, this is because of the object-oriented nature of GUIs. Inherent to GUIs are side-effects, which are necessary to show components or to handle events. Therefore, most people turn to an object-oriented language to build a GUI. However, Haskell provides a number of features that aid in building GUIs. Side-effects can be modelled relatively easily using monads. In addition, features like polymorphism, higher-order functions, and type classes improve the possibility to abstract over common patterns.

Additionally, a prejudice exists that most applications implemented in Haskell do not even require a GUI. However, there are many examples of Haskell applications that do have a GUI. These examples include an editor for Bayesian belief networks called Dazzle [14], a graphical heap profile viewer for GHC [25] called HPView, a document-centered presentation of Haskell called Pivotal [15], and a generic structure editor called Proxima [24]. Even a first-person shooter called Frag [10] has been implemented in Haskell. In fact, more and more Haskell applications are written that offer a GUI.

In this paper, we classify the libraries for building GUIs in Haskell in two categories: monadic and declarative. A monadic library gives a more imperative feel to building GUIs by allowing the user to bind variables, often in the *IO* monad. This style is closer to object-oriented programming than a declarative library. A declarative library uses point-free style in combination with an abstraction of monads, called arrows [17]. The point-free style forces the user to use library combinators to build the GUI. Because of these combinators, it is hard to specify the static appearance of a GUI (i.e., creating and laying out components). However, specifying the dynamic behaviour of a GUI (i.e., event handling)

\*This work is carried out in the context of the Software Technology Colloquium at the Universiteit Utrecht

is easier in a declarative library because arguments are implicitly passed to the right components by using the library combinators. Since the largest part of building GUIs is concerned with the static appearance of components, building GUIs with declarative libraries is hard. Furthermore, most people have minimal knowledge of arrows. Therefore, this paper discusses two popular monadic libraries: Gtk2Hs [8] and wxHaskell [19]. A running example is used to compare these libraries and to answer the question: what determines the usability of a GUI library in Haskell?

Section 2 introduces Gtk2Hs based on the implementation of the running example. After that, the implementation of the running example using wxHaskell is discussed in Section 3. Following that, the main differences between both libraries are discussed in Section 4. Section 5 gives a short overview of implementation issues of GUI libraries. Finally, Section 6 discusses related work and Section 7 ends with some concluding remarks.

### 2. GTK2HS

We start by exploring the Gtk2Hs library. The running example we discuss is a small application (dubbed Currency Converter) for converting values between two currencies: the euro and the guilder. The GUI consists of an input field in which the user can enter a value, and a button which converts the value, as can be seen in Figure 1. Clicking the button changes the value of the input field and the label associated with the button. This example has been chosen because it is minimal and contains the typical aspects of building a GUI: managing state, managing widgets, specifying layout, and dealing with events. Figure 2 contains code that is independent of the library being used.

Additionally, we discuss the architecture and main features of the Gtk2Hs library. A specific powerful feature of Gtk2Hs called Glade, is further explored at the end of this section.

#### 2.1 Example

Consider the following definition of *main*, which constructs the complete GUI.

```
main :: IO ()
main =
  do initGUI

     var ← newIORef (2.20371, Euro)

     frm ← windowNew
```



(a) Gtk2Hs



(b) wxHaskell

Figure 1: Screenshots of the Currency Converter

```

windowSetTitle frm "Converter"

ent ← entryNew
entrySetText ent "0"

btn ← buttonNew
buttonSetLabel btn (toLabel Guilder)

box ← hboxNew True 5
set box [containerChild := ent]
set box [containerChild := btn]
set frm [containerChild := box]

let click = change var ent btn
onClicked btn click

widgetShowAll frm
mainGUI

```

First, we call the function *initGUI* to initialize the GUI toolkit. In order to model state, a mutable variable is created using the function *newIORef*. The initial value is passed to the function creating the mutable variable. Next, a window, an input field, and a button are created. Properties associated with these widgets have to be set after the widget is created. There are two distinct possibilities in setting the properties of a widget. Other than using functions specific to a widget and its attribute (e.g. *buttonSetLabel*), the function *set* can be used. This function takes a list of properties which are set on the widget given to the function. There is no special syntax (i.e., the *:=* operator) involved in setting the properties; this is actually an infix constructor. The exact implementation details of this feature are beyond the scope of this paper and can be found elsewhere [8, 19].

After the widgets have been created, the layout of the application can be specified by defining the hierarchical structure between widgets. In order to add a widget as a child to another widget (e.g., a button to a frame), the child widgets are wrapped in a special layout component, called *box*. Layout components are convenient in the sense that these components take care of determining the size of its children. Even if the size of the frame changes at runtime (e.g., when the user maximizes the frame), the layout component dynamically changes the sizes of its children according to the properties of the layout component.

```

data Currency = Euro | Guilder
deriving Show

```

```

toLabel :: Currency → String
toLabel = ("To " ++) . show

```

```

type State = (Float, Currency)

```

```

flipState :: State → State
flipState (rate, Euro) = (1 / rate, Guilder)
flipState (rate, Guilder) = (1 / rate, Euro)

```

Figure 2: Library-independent code

For our running example, a layout component is created that aligns its children horizontally. Applying the function *hboxNew* to the arguments *True* and *5* results in a horizontal box which gives equal space to its children, and adds spacing of five pixels between each child. Each child of the horizontal box is added by setting the *containerChild* attribute of the horizontal box. Since this attribute is set successively for the input field and the box, it looks like this attribute is redefined. However, the *get* and *set* functions of a widget can have different types in the Gtk2Hs library. For example, setting the *containerChild* attribute is actually an update of the list of children of the horizontal box. As a consequence, retrieving the value of this attribute results in a list of children. After the input field and the button are added to the horizontal box, the layout component itself can be added to the frame.

Now that the static aspect of the GUI is finished, the dynamic aspect of the GUI is implemented. A callback function is attached to the click event of the button. Each component that must change when the event is triggered is passed explicitly to the callback function.

Finally, by calling *widgetShowAll*, the main frame and all of its children are shown. The main event loop of Gtk2Hs is started by calling *mainGUI*. This event loop intercepts events triggered by the user and calls the appropriate callback functions related to the intercepted event.

As mentioned before, the arguments necessary to handle the triggered event properly are passed to the callback function *change* explicitly. When the user clicks on the button, the current state, the input field, and the label of the button must change.

```

type StateVar = IORef State

```

```

change :: StateVar → Entry → Button → IO ()
change var ent btn =

```

```

do state@(rate, curr) ← readIORef var
input ← entryGetText ent
case string2float input of
Just x → do writeIORef var (flipState state)
            entrySetText ent (show $ rate * x)
            buttonSetLabel btn (toLabel curr)
Nothing → return ()

```

```

string2float :: String → Maybe Float
string2float string = case reads string of
    [(result, "")] → Just result
    _                → Nothing

```

When the button is clicked, the current state is read from the mutable variable and the value of the input field is retrieved. It is important to realize that the value itself is of type *String*. Therefore, the value is converted explicitly using the conversion function *string2float*. When this conversion fails, nothing has to be done. Otherwise, a new state is written to the mutable variable. Following that, the new value is calculated and written to the input field. Finally, the new label of the button is set according to the old state.

## 2.2 Architecture

The Gtk2Hs library is built on top of GTK+ [2]. This is a large open source toolkit containing many widgets, which is implemented in C. For each widget supported by the Gtk2Hs API, a function should be written that calls the corresponding function in C using the Foreign Function Interface (FFI) [9]. Since a lot of functions are available, writing each function by hand would be quite tiresome. Therefore, a code generator was written, which generates the appropriate functions according to the implementation in C. Another advantage of this is that almost the full capabilities of GTK+ can be used in Gtk2Hs. Additionally, a generated API is more maintainable since changes in the underlying toolkit can be reflected in the other API quite easily by running the code generator again.

Besides almost covering the complete GTK+ API, Gtk2Hs is platform independent. Furthermore, bindings exist for the Cairo vector graphics library and several Gnome modules. The Gnome modules which are currently supported are GConf for storing application preferences, and SourceView for syntax highlighting in a text editor. Additionally, it is possible to write a browser in Haskell since there is a binding for the Mozilla browser rendering engine available. Some less thrilling features of Gtk2Hs are its extensive documentation and unicode support.

## 2.3 Glade

One of the hardest parts of building GUIs is implementing the layout using library functions. First of all, it is not intuitive to specify a layout by using library functions. Moreover, these library functions usually have several arguments which specify the properties of the layout component. Compiling the code is the only way to view the effect of changing these properties. This could be quite tiresome when building a complex GUI, since the immediate effect of a property is not visible.

Therefore, Gtk2Hs provides another approach for specifying a layout: the visual GUI builder Glade [1], which comes with GTK+. This application allows the user to drag 'n drop widgets. Additionally, it is now possible to view the immediate effect of changing a property of a widget. When the desired layout is specified, the specification can be exported to a file in XML format. Consider the following XML fragment exported by Glade for the layout specification of the Currency Converter.

```

...
<child>
  <widget class="GtkHBox" id="box">
    <property name="visible">True</property>
    <property name="homogeneous">False</property>
    <property name="spacing">0</property>

    <child>
      <widget class="GtkEntry" id="ent">
        ...
      </child>
    ...
  </child>
...

```

The user must assign unique identifiers to each widget, which are defined as the *id* attribute in the above XML snippet. Passing such an identifier to the function *xmlGetWidget* allows the user to extract the right widget from the XML file. Using this approach, the creation and specification of widgets is done using Glade. This means that it is not necessary any more to set the properties of a widget once it has been extracted from the XML file.

```

main :: IO ()
main =
  do initGUI

     Just xml ← xmlNew "converter.glade"

     var ← newIORef (2.20371, Euro)

     frm ← xmlGetWidget xml castToWindow "frm"
     ent ← xmlGetWidget xml castToEntry  "ent"
     btn ← xmlGetWidget xml castToButton "btn"

     let click = change var ent btn
         onClicked btn click

     widgetShowAll frm
     mainGUI

```

This new definition of *main* closely resembles the previous definition of *main*. The most significant difference is that widgets are no longer created manually but loaded from the XML definition of the layout. As can be seen in the example, a function is passed to cast the widget to the right type. However, this does not seem necessary since all the information is available in the XML file. The library could have implemented a function mapping the type of a widget to the appropriate cast function (e.g., mapping "GtkEntry" to the function *castToEntry*).

Since the children of the frame are already defined using Glade, retrieving this widget from the specification would suffice to show the layout. However, the other widgets need to be passed explicitly to the callback function. Therefore, these widgets are still retrieved explicitly. Note that the definition of the event handler *change* stays the same.

## 3. WXHASKELL

This section discusses the implementation of the Currency Converter using wxHaskell. Again, we will discuss the architecture and the main features. Section 3.3 discusses an extension to the wxHaskell library, called the XTC library.

### 3.1 Example

The GUI for the Currency Converter is constructed by the following definition of *main*.

```
main :: IO ()
main = start $
  do var ← variable [value := (2.20371, Euro)]

     frm ← frame [text := "Converter"]

     ent ← mkValueEntry frm [typedValue := Just 0]

     btn ← button frm [text := toLabel Guilder]

     set frm [layout := row 5 [widget ent, widget btn]]

  let click = change var ent btn
      set btn [on command := click]
```

The only administrative task that is necessary is calling the function *start* with as argument the constructed GUI. Next, a mutable variable containing the state of the application is created with its initial value. The library abstracts over mutable variables using the function *variable*. The implementation itself uses *IORef a*, but because of this abstraction the user is not confronted with this detail.

After creating the mutable variable, the widgets of the application are created. The functions available for creating a widget are consistent in the order in which they expect their arguments. The *first* argument of such a function is always the parent widget, if there is one. The *last* argument is a list of properties which are immediately set on the widget which is created. This approach differs from the approach taken by the *Gtk2Hs* library where setting the parent widget, and properties of a widget, is done after the widget is created.

The *wxHaskell* library implements setting and getting the value of properties using a similar approach to *Gtk2Hs*, that is, using the `:=` infix constructor. However, *wxHaskell* implements properties in a more generic fashion than the *Gtk2Hs* library. As can be seen from the definition of *main*, the frame widget and button widget have a similar attribute called *text*. With *Gtk2Hs*, setting this property is implemented for a frame and a button separately. The *wxHaskell* library solves this problem by combining type classes with phantom types, which will be discussed in Section 3.2.

The layout is implemented using layout combinators, similar to the functions available in the *Gtk2Hs* library. The function *row* is similar to the *hbox* component from the implementation using *Gtk2Hs*. Again, a spacing of five pixels is desired between each child of the frame. In contrast to the *hbox* component, a number of widgets are added in one call instead of a call for each widget. Note that it is necessary to upcast both *ent* and *btn* using *widget* to a layout component, in order to assign this value to the *layout* attribute. Moreover, upcasting is necessary to put these different widgets together in a list.

Another difference compared to the implementation using *Gtk2Hs* is that the input field widget is typed. A special library provides this abstraction, which guarantees that the value retrieved from the input field is correctly typed. Note that the initial value is set to *Just 0*. This is because the

types of the values in the *get* and *set* function are the same. Since parsing can fail, the type of the value that is retrieved from the attribute is *Maybe Float*. Therefore, setting the value of the input field also requires a value of this type. This extension to the *wxHaskell* library is discussed in Section 3.3.

Finally, the callback function is defined and attached to the event associated with the button. Again, every component that is needed to handle the triggered event properly, is passed explicitly to the callback function.

**type** *StateVar* = *Var State*

```
change :: StateVar      →
        ValueEntry Float () →
        Button ()       → IO ()
change var ent btn =
  do (rate, curr) ← get var value
     input ← get ent typedValue
     case input of
       Just x  → do set var [value ∼ flipState]
                    set ent [typedValue := Just (rate * x)]
                    set btn [text := toLabel curr]
       Nothing → return ()
```

As can be seen from the type signature, the second argument is the typed input field. Retrieving the input from the user results in a value of type *Maybe Float*. This means that no explicit conversion is necessary: this is taken care of by the *XTC* library. In case the input can be parsed, the current state and the widgets involved are changed. First, the current state is flipped by setting the value attribute of the variable. The `∼` notation provides a shorthand for modifying the value. Then, the value of the input field is set and the label of the button is changed.

### 3.2 Architecture

The *wxHaskell* library is built on top of *wxWidgets* [4], which itself is implemented using *GTK+*. In this way, the *wxWidgets* library can provide more functionality than just *GTK+* itself. The main features of *wxHaskell* are the same as *Gtk2Hs*: platform independence, extensive documentation and unicode support.

The *wxHaskell* library actually consists of two layers: *WX* and *WXCore*. The latter provides functions to the *wxWidgets* library using FFI, and it is generated from the *wxWidgets* source for the same reasons the *Gtk2Hs* API is generated. The additional *WX* layer provides a number of abstractions to the *WXCore* layer, which improves the usability of the complete library. We will highlight two specific features from this abstractions layer: inheritance between widgets and shared attributes.

#### *Inheritance between widgets*

Because of the object-oriented nature of the underlying toolkit, the inheritance relation must also be reflected in the abstraction layer. We start by defining a type synonym for an object, which is a pointer to a C++ object.

**type** *Object a* = *Ptr a*

Next, *phantom data types* are introduced for each C++ class. These data types are special in the sense that no con-

structors are present at all. Consider the following phantom data types for the classes *Window*, *Control*, and *Button*.

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

These data types are parameterized with a single type variable, which is used to model the inheritance relation.

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

In this way, *Object* becomes a superclass of *Window*, which is a superclass of *Control*, which is a superclass of *Button*. These relations become apparent if we unfold the type synonym *Button* stepwise.

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

Using the approach taken by the wxHaskell library, phantom types with a single type variable, it is not possible to define multiple inheritance. Consider the following type synonyms which model that *Window* is a subclass of both *Object* and *OtherObject*.

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

This is not valid Haskell because the type synonym for *Window* is duplicated. Therefore, it is not possible to model multiple inheritance using this approach. However, it is yet unknown if a different approach using phantom types does allow multiple inheritance.

The notion of (single) inheritance can be used in the type of a function to distinguish between instances. Consider the type of the function that creates a button.

```
button :: Window a → [Prop (Button ())] → IO (Button ())
```

This function expects *at least an instance* of *Window* as its first argument. The button returned by the function is an *exact* instance of *Button*, because of the `()` in the type which prevents further nesting of types.

This approach of modelling the inheritance relation is more powerful than using type classes, because it allows to distinguish between instances. Type classes are used by the Gtk2Hs library to model inheritance, although only cast functions are part of these type classes.

### Shared attributes

As mentioned before, different widgets can have the same attribute. For example, both the frame widget and the button widget have the attribute *text*. An obvious solution would be to define this attribute as a function.

```
text :: Attr (Window a) String
```

However, this solution is not satisfactory because there is no uniform implementation available for the text attribute. Therefore, combining inheritance with overloading results in the best solution: it allows us to define the attribute specific to a widget.

```
class Textual w where
    text :: Attr w String
```

```
instance Textual (Button a) where
    text = ...
```

This approach still allows the user to define their own widgets, and profit from the shared attribute. The user is obligated to provide an instance of *Textual* to be able to use the *text* attribute with their own widget.

## 3.3 XTC

In general, input entered by the user is retrieved as a value of type *String*. Usually, this is not desirable and the value must be converted explicitly every time the value is used. The eXtended and Typed Control (XTC) library [14, 19] solves this inconvenience by providing typed widgets. In the case of an input field, the user can choose what the type should be of the input entered by the user. If there are instances available for the *Read* and *Show* type classes, the library takes care of parsing and showing the value. In case parsing fails, the library automatically provides visual feedback to the user.

Another example of a typed widget is the typed radio button group. It is often desirable to know which radio button is selected by the user. Normally, this would return an index with which the accompanying value can be found using a lookup function. The XTC library solves this by performing this lookup for you. This means that if the selected radio button is retrieved from the typed radio button group, the corresponding value is returned.

## 4. COMPARISON

We have seen two implementations of the running example, one using Gtk2Hs, and one using wxHaskell. The strongest points of both libraries are discussed, together with a setting in which these strong points will be most useful.

### Gtk2Hs

As discussed earlier, it is desirable to design the layout of a GUI visually. Glade is of great help in this process because it shows the effect of adding widgets and setting properties immediately. When the layout is implemented by calling specific functions, compilation of the code is necessary to view the changes made. Since laying out components is not easy, especially with large applications, this process is quite tedious. Therefore, this feature of Gtk2Hs helps in designing layout for complex and large applications. Unfortunately, the wxHaskell library does not provide any kind of visual GUI builder and forces the user to use layout combinators. However, there are possibilities for wxHaskell to use Glade as well since Glade is not tied to Gtk2Hs.

The implementation of loading the layout specification exported by Glade is not yet optimal. A disadvantage of loading a layout specification at runtime is that this specification

must be deployed with the application itself. Since the layout specification is plain text, the layout could be changed by the user, which is not desirable. An advantage of loading the specification at runtime is that changing the layout does not require compiling the application again. An alternative approach would be to generate code according to the layout specification.

Another strong point of the Gtk2Hs library is the fact that a binding for the Cairo vector graphics library exists, which increases the graphical power. The wxHaskell library has no additional bindings for graphical libraries, but this does not seem impossible to implement.

### *wxHaskell*

In general, the wxHaskell library provides more abstractions than the Gtk2Hs library. First of all, a user building a GUI wants to perform as less administrative tasks as possible, but still wants to have the possibility to perform these tasks manually. Both libraries provide functions for these tasks, but differ in the abstractions provided for these tasks. For example, Gtk2Hs requires the user to initialize the GUI toolkit and start the main event loop manually. This is in contrast with the wxHaskell library which requires a single function call to start up the complete process. It does not seem hard to define such a function for Gtk2Hs, nevertheless, it is still missing:

```
start :: WidgetClass w => IO w -> IO ()
start gui = do initGUI
              wdg ← gui
              widgetShowAll wdg
              mainGUI
```

A powerful feature of wxHaskell is the way in which properties are implemented. The library provides nice abstractions to manage the attributes of a widget. Additionally, the library implements shared attributes, which allows programmers to use the same attribute on different widgets. Unfortunately, the Gtk2Hs library contains a specific function for each attribute and widget.

The wxHaskell library nicely implements inheritance using phantom types, which has a more object-oriented feel to it and enforces the fact that wxHaskell is an imperative library. The Gtk2Hs library has implemented inheritance between widgets using type classes, which is less powerful than using phantom types. The approach used by Gtk2Hs does not allow to distinguish between different instances.

Another disadvantage of the Gtk2Hs library is that the parent of a widget is set after the widget has been created. Since this is done frequently, it would have been really useful to add an argument to the function creating the widget. This argument should have denoted the parent widget, similar to the wxHaskell library. This also prevents that children have multiple parents, which can result in unexpected behaviour.

Finally, the XTC extension of wxHaskell greatly improves the usability of the library. The advantage of using typed input fields is that verification of the input is done by the library. Therefore, there is no need for a conversion function as with Gtk2Hs. A small drawback of using the XTC library is that setting the value of a typed widget involves a value

of type *Maybe a*. This means that it is possible to set the initial value to *Nothing*, which does not really seem useful. A useful feature would be to have independent types for the *set* and *get* functions, as with Gtk2Hs. This would allow to set a value of type *a*, and retrieve a value of type *Maybe a*.

## 5. LIBRARY IMPLEMENTATION

We have seen two monadic libraries, but there are a number of other libraries as well. A common property of these libraries is that none of these are built from scratch, they are built on top of an existing toolkit. This is easier than implementing your own library, because a lot of work is already done. Ignoring all the effort put into the existing libraries is senseless. Moreover, your own library can profit from the success of the underlying toolkit.

### 5.1 Garbage collection

Most existing libraries are implemented in a language other than Haskell. Therefore, the FFI is used to interact with other languages, mostly C and C++ in the case of GUI libraries. The FFI introduces two major problems which should be taken care of by a GUI library. Both problems are concerned with (the absence of) garbage collection. Keep in mind that there is a garbage collector for Haskell, while the garbage collector for C/C++ is absent.

#### *Haskell value referencing C/C++ value*

Because there is no garbage collector for C/C++, these C/C++ values must be garbage collected explicitly. Since the Haskell value is the only active reference, this Haskell value is responsible for the garbage collection of the C/C++ value. When the Haskell value is garbage collected, the C/C++ value must be garbage collected as well. Therefore, the Haskell garbage collector must call the appropriate C/C++ function to clean up the garbage. In order to call such a function at the right time, a reference scheme must be used to keep track of the number of active references. A foreign pointer (*ForeignPtr*) implements such a reference scheme and is usually used to solve the problems discussed.

#### *C/C++ value referencing Haskell value*

The main problem is that the Haskell garbage collector is unaware of references from C/C++ values. As a consequence, it could be the case that a Haskell value is garbage collected while there are still active references from C/C++ values. Additionally, the garbage collector could move the Haskell value around, which also can result in invalid references. A common approach to solving these problems is by using a special pointer for Haskell values being referenced by C/C++ values. Such a special pointer is called a stable pointer (*StablePtr*) and is stored in a special table. Stable pointers stored in this table are treated differently from normal pointers: a stable pointer is untouched by the Haskell garbage collector. Although this approach is safe, it is not the most efficient approach: stable pointers stay active until the application terminates.

### 5.2 Concurrency

Concurrency is another implementation issue that GUI libraries have to deal with. If the GUI of an application runs in the same thread as the application itself, the user can notice delay in using the GUI. Therefore, the GUI thread must be different from the application thread. In contrast

with the next Haskell standard, the current Haskell standard does not support a concurrency model. Therefore, Gtk2Hs, wxHaskell, and most other GUI libraries do not support multiple threads.

## 6. RELATED WORK

Besides Gtk2Hs and wxHaskell, there are a number of libraries available for building GUIs in Haskell. However, most of these libraries are experimental and are not well maintained. We discuss these other libraries grouped by the underlying toolkit.

One of the first libraries used to build GUIs in Haskell is the Fudget library [7]. This declarative library consists of separate components, called fudgets, which are connected using wires. With these wires, fudgets communicate with each other using message passing. Based on this concept, Gadgets [20] were developed as an extension to Fudgets by using concurrent computations in contrast to the sequential computations of Fudgets. These two libraries are built upon the X Window System [5]. This platform independent system consists of an X server which accepts requests to provide graphical output, and sends back events on input from a user. Another declarative library based on this system is Haggis [13]. This library provides a multi-threaded, concurrent user interface which is built on top of the concurrent extension of GHC. Concurrency is used by the library to structure interaction with the user.

Another way of handling interaction with the user is by means of Functional Reactive Programming (FRP) [16]. The key concepts of FRP are behaviour and events, in which behaviour changes a value over time. The FRP paradigm is further extended to the Arrowized Functional Reactive Programming (AFRP) paradigm, which uses arrow combinators and arrow notation. This paradigm extends the concepts of FRP to signals and signal transformers. The first library to use this paradigm is called Fran [26] which uses DirectX for its graphical output. Other implementations using the AFRP paradigm are Fruit [11] and Yampa [12]. The former uses a vector graphics library named Heaven for graphical output, while the latter uses the Haskell Graphics Library for its graphical output. All three libraries mentioned have in common that they are based on the (A)FRP paradigm. Using the definition of styles mentioned before, these libraries can be considered declarative libraries.

Several imperative libraries are based on a system called Tcl/Tk [3]. This system is a combination of the Tcl scripting language and the Tk widget toolkit. This system is platform independent and provides a nice library of standard widgets available for building GUIs. One of the libraries built on top of this system is called TclHaskell [22]. This library provides nice abstractions for building GUIs in Haskell. However, to provide more power and to be even more declarative, an extension called FranTk [23] is implemented. Another library that uses the Tcl/Tk system is called Htk [18]. This library only uses arrow notation for event handling. However, the main style of this library can be considered imperative.

HToolkit [6] is a library which uses GTK+, as well as Gtk2Hs. The HToolkit library uses several low-level libraries to communicate with GTK+. It provides several abstractions in an imperative style to build a GUI in Haskell.

A competitor of GTK+ is wxWidgets [4], which provides an extensive amount of widgets as well. As mentioned earlier, the monadic library wxHaskell is built on top of wxWidgets. A library that tries to combine the imperative style of wxHaskell with the declarative style of Fruit is wxFruit [21]. However, this library is experimental and does not yet fulfill its goal of combining the best of both styles.

## 7. CONCLUSION

This paper discusses two monadic libraries, and compares these using a running example. These libraries have been compared particularly on their usability. It turns out that the abstractions provided determine the usability. These abstractions should be clear, intuitive, and not too high-level. The Gtk2Hs library lacks this property because hardly any useful abstractions are available. On the other hand, wxHaskell does provide a number of useful abstractions, which greatly improves the usability of the library.

As we have seen, cross-pollination occurs between Gtk2Hs and wxHaskell. Features developed by one library can often be found in the other library. There are still a number of features that have to be implemented by either Gtk2Hs or wxHaskell. If both libraries keep profiting from each other, they will both increase in usability. Cross-pollination will improve the quality of these and other libraries in the near future.

## 8. REFERENCES

- [1] The Glade GUI builder. <http://glade.gnome.org>.
- [2] GTK+. <http://www.gtk.org>.
- [3] The Tcl/Tk library. <http://www.tcl.tk>.
- [4] The wxWidgets library. <http://www.wxwidgets.org>.
- [5] The X Window System. <http://www.x.org>.
- [6] Krasimir Angelov. The HToolkit project. <http://hToolkit.sourceforge.net>.
- [7] Magnus Carlsson and Thomas Hallgren. Fudgets: a graphical user interface in a lazy functional language. In *FPCA '93: Proceedings of the ACM SIGPLAN 1993 conference on Functional Programming Languages and Computer Architecture*, pages 321–330, New York, NY, USA, 1993. ACM Press.
- [8] Manuel Chakravarty, Duncan Coutts, and Axel Simons. The Gtk2Hs library. <http://haskell.org/gtk2hs>.
- [9] Manuel Chakravarty *et al.* The Haskell 98 foreign function interface 1.0: an addendum to the Haskell 98 report. <http://www.cse.unsw.edu.au/~chack/haskell/ffi>.
- [10] Mun Hon Cheong. Frag. <http://haskell.org/haskellwiki/Frag>.
- [11] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell '01: Proceedings of the 2001 ACM SIGPLAN workshop on Haskell*, pages 41–69, New York, NY, USA, 2001. ACM Press.

- [12] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18, New York, NY, USA, 2003. ACM Press.
- [13] Sigbjorn Finne and Simon Peyton Jones. Composing the user interface with Haggis. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 1–37, London, UK, 1996. Springer-Verlag.
- [14] Linda van der Gaag, Arjan van IJzendoorn, and Martijn Schrage. Haskell ready to dazzle the real world. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 17–26, New York, NY, USA, 2005. ACM Press.
- [15] Keith Hanna. A document-centered environment for haskell. In *IFL '05: Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages*, 2005.
- [16] Paul Hudak and Zhanyong Wan. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, pages 242–252, New York, NY, USA, 2000. ACM Press.
- [17] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.
- [18] Einar Karlsen, Andree Lütke, Christoph Lüth, and George Russell. The HTk project  
<http://www.informatik.uni-bremen.de/htk>.
- [19] Daan Leijen. wxHaskell: a portable and concise GUI library for Haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, New York, NY, USA, 2004. ACM Press.
- [20] Rob Noble and Colin Runciman. Gadgets: Lazy functional components for graphical user interfaces. In *PLILPS '95: Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 321–340, London, UK, 1995. Springer-Verlag.
- [21] Bart Robinson. wxFruit: A practical GUI toolkit for functional reactive programming.  
<http://zoo.cs.yale.edu/classes/cs490/03-04b/bartholomew.robinson>.
- [22] Meurig Sage. The TclHaskell project.  
<http://www.dcs.gla.ac.uk/~meurig/TclHaskell>.
- [23] Meurig Sage. FranTk - a declarative GUI language for Haskell. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 106–117, New York, NY, USA, 2000. ACM Press.
- [24] Martijn Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, October 2004.
- [25] Wei Tan. The HPVView project.  
<http://dready.org/projects/HPVView>.
- [26] Simon Thompson. A functional reactive animation of a lift using Fran. *Journal of Functional Programming*, 10(3):245–268, 2000.