wxHaskell

A Portable and Concise GUI Library for Haskell

Daan Leijen
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
daan@cs.uu.nl

Abstract

wxHaskell is a graphical user interface (GUI) library for Haskell that is built on wxWidgets: a free industrial strength GUI library for C++ that has been ported to all major platforms, including Windows, Gtk, and MacOS X. In contrast with many other libraries, wxWidgets retains the native look-and-feel of each particular platform. We show how distinctive features of Haskell, like parametric polymorphism, higher-order functions, and first-class computations, can be used to present a concise and elegant monadic interface for portable GUI programs.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (Functional) Programming*; D.2.2 [**Design Tools and Techniques**]: User interfaces.

General Terms

Design, Languages.

Keywords

Graphical user interface, combinator library, layout, wxWidgets, Haskell, C++.

1 Introduction

The ideal graphical user interface (GUI) library is efficient, portable across platforms, retains a native look-and-feel, and provides a lot of standard functionality. A Haskell programmer also expects good abstraction facilities and a strong type discipline. wxHaskell is a free GUI library for Haskell that aims to satisfy these criteria [25].

Haskell'04, September 22, 2004, Snowbird, Utah, USA.

There is no intrinsic difficulty in implementing a GUI library that provides the above features. However, the amount of work and maintainance associated with such project should not be underestimated; many GUI libraries had a promising start, but failed to be maintained when new features or platforms arose. With wxHaskell, we try to avoid this pitfall by building on an existing cross-platform framework named wxWidgets: a free industrial strength GUI library for C++ [43].

wxWidgets provides a common interface to native widgets on all major GUI platforms, including Windows, Gtk, and Mac OS X. It has been in development since 1992 and has a very active development community. The library also has strong support from the industry and has been used for large commercial applications, for example, AOL communicator and AVG anti-virus.

wxHaskell consists of two libraries, *WXCore* and *WX*. The *WXCore* library provides the core interface to wxWidgets functionality. It exposes about 2800 methods and more than 500 classes of wxWidgets. Using this library is just like programming wxWidgets in C++ and provides the raw functionality of wxWidgets. The extensive interface made it possible to already develop substantial GUI programs in wxHaskell, including a Bayesian belief network editor and a generic structure editor, called Proxima [41]. The *WXCore* library is fully Haskell'98 compliant and uses the standard foreign function interface [11] to link with the wxWidgets library.

The *WX* library is implemented on top of *WXCore* and provides many useful functional abstractions to make the raw wxWidgets interface easier to use. This is where Haskell shines, and we use type class overloading, higher-order functions, and polymorphism to capture common programming patterns. In particular, in Section 6 we show how *attribute abstractions* can be used to model widget settings and event handlers. Furthermore, *WX* contains a rich combinator library to specify layout. In section 7, we use the layout combinator library as a particular example of a general technique for declarative abstraction over imperative interfaces. As described later in this article, the *WX* library does use some extensions to Haskell'98, like existential types. Most of this article is devoted to programming wxHaskell with the *WX* library, and we start with two examples that should give a good impression of the functionality of the library.

2 Examples

Figure 1 is a small program in wxHaskell that shows a frame with a centered label above two buttons. Pressing the *ok* button closes the frame, pressing the *cancel* button changes the text of the label.

```
main = start gui
gui :: IO ()
gui =
  do f   ← frame    [text := "Example"]
     lab ← label  f [text := "Hello wxHaskell"]
     ok  ← button f [text := "Ok"]
     can ← button f [text := "Cancel"]
     set ok  [on command := close f]
     set can [on command := set lab [text := "Goodbye?"]]
     set f [layout := column 5 [floatCenter (widget lab)
                               ,floatCenter $
                                row 5 [widget ok, widget can]]]
```

Figure 1. A first program in wxHaskell, with screenshots on Windows XP and Linux (Gtk)

A graphical wxHaskell program[1] is initialized with the *start* function, that registers the application with the graphical subsystem and starts an event loop. The argument of *start* is an *IO* value that is invoked at the initialization event. This computation should create the initial interface and install further event handlers. While the event loop is active, Haskell is only invoked via these event handlers.

The *gui* value creates the initial interface. The *frame* creates a top level window frame, with the text `"Example"` in the title bar. Inside the frame, we create a text label and two buttons. The expression *on command* designates the event handler that is called when a button is pressed. The *ok* button closes the frame, which terminates the application, and the *cancel* button changes the content of the text label.

Finally, the *layout* of the frame is specified. wxHaskell has a rich layout combinator library that is discussed in more detail in Section 7. The text label and buttons float to the center of its display area, and the buttons are placed next to each other. In contrast to many dialogs and windows in contemporary applications, this layout is fully resizeable.

3 Asteroids

We now discuss a somewhat more realistic example of programming with wxHaskell. In particular we look at a minimal version of the well known *asteroids* game, where a spaceship tries to fly through an asteroid field. Figure 2 shows a screenshot. Even though we use a game as an example here, we stress that wxHaskell is a library designed foremost for user interfaces, not games. Nevertheless, simple games like asteroids show many interesting aspects of wxHaskell.

The game consists of a spaceship that can move to the left and right using the arrow keys. There is an infinite supply of random rocks (asteroids) that move vertically downwards. Whenever the spaceship hits a rock, the rock becomes a flaming ball. In a more realistic version, this would destroy the ship, but we choose a more peaceful variant here. We start by defining some constants:

```
height   = 300
width    = 300
diameter = 24
chance   = 0.1 :: Double
```

[1] One can access wxHaskell functionality, like the portable database binding, without using the GUI functionality.

For simplicity, we use fixed dimensions for the game field, given by *width* and *height*. The *diameter* is the diameter of the rocks, and the *chance* is the chance that a new rock appears in a given time frame. The main function of our game is *asteroids* that creates the user interface:

```
asteroids :: IO ()
asteroids =
  do g      ← getStdGen
     vrocks ← variable [value := randomRocks g]
     vship  ← variable [value := div width 2]
     f      ← frame    [resizeable := False]
     t      ← timer f  [interval := 50
                       ,on command := advance vrocks f]
     set f [text         := "Asteroids"
           ,bgcolor      := white
           ,layout       := space width height
           ,on paint     := draw vrocks vship
           ,on leftKey   := set vship [value :~ \x → x-5]
           ,on rightKey  := set vship [value :~ \x → x+5]
           ]
```

First a random number generator *g* is created that is used to randomly create rocks. We create two mutable variables: *vrocks* holds an infinite list that contains the positions of all future rock positions, *vship* contains the current *x* position of the ship.

Next, we create the main window frame *f*. The frame is not resizeable, and we can see in the screenshot that the maximize box is greyed out. We also attach an (invisible) timer *t* to this frame that ticks every 50 milliseconds. On each tick, it calls the function *advance* that advances all rocks to their next position and updates the screen.

Finally, we set a host of attributes on the frame *f*. Note that we could have set all of these immediately when creating the frame but the author liked this layout better. The text *Asteroids* is displayed in the title bar, and the background color of the frame is white. As there are no child widgets, the *layout* just consists of empty space of a fixed size. The attributes prefixed with *on* designate event handlers. The *paint* event handler is called when a redraw of the frame is necessary, and it invokes the *draw* function that we define later in this section. Pressing the left arrow key or right arrow key changes the x position of the spaceship. In contrast to the $(:=)$ operator, the $(:\sim)$ operator does not assign a new value, but applies a function to the attribute value, in this case, a function that increases or

decreases the x position by 5 pixels. A somewhat better definition would respect the bounds of the game too, for example:

```
on leftKey := set vship [value :~ \x → max 0 (x − 5)]
```

The *vrocks* variable holds an infinite list of all future rock positions. This infinite list is generated by the *randomRocks* function that takes a random number generator *g* as its argument:

```
randomRocks :: RandomGen g ⇒ g → [[Point]]
randomRocks g =
  flatten [] (map fresh (randoms g))
flatten rocks (t : ts) =
  let now  = map head rocks
      later = filter (not ∘ null) (map tail rocks)
  in now : flatten (t ++ later) ts
fresh r
  | r > chance = []
  | otherwise  = [track (floor (fromIntegral width ∗ r / chance))]
track x =
  [point x (y − diameter) | y ← [0, 6 .. height + 2 ∗ diameter]]
```

The standard *randoms* function generates an infinite list of random numbers in the range $[0,1)$. The *fresh* function compares each number agains the *chance*, and if a new rock should appear, it generates a finite list of positions that move the rock from the top to the bottom of the game field. The expression *map fresh* (*randoms g*) denotes an infinite list, where each element contains either an empty list, or a list of positions for a new rock. Finally, we *flatten* this list into a list of time frames, where each element contains the position of every rock in that particular time frame.

The *advance* function is the driving force behind the game, and it is called on every timer tick.

```
advance vrocks f =
  do set vrocks [value :~ tail]
     repaint f
```

The *advance* function advances to the next time frame by taking the *tail* of the list. It then forces the frame *f* to repaint itself. The *paint* event handler of the frame calls the *draw* function that repaints the game:

```
draw vrocks vship dc view =
  do rocks ← get vrocks value
     x     ← get vship value
     let ship      = point x (height − 2 ∗ diameter)
         positions = head rocks
         collisions = map (collide ship) positions
     drawShip dc ship
     mapM_ (drawRock dc) (zip positions collisions)
     when (or collisions) (play explode)
```

The *draw* function was partially parameterised with the *vrocks* and *vship* variables. The last two parameters are supplied by the paint event handler: the current *device context* (*dc*) and view area (*view*). The device context is in this case the window area on the screen, but it could also be a printer or bitmap for example.

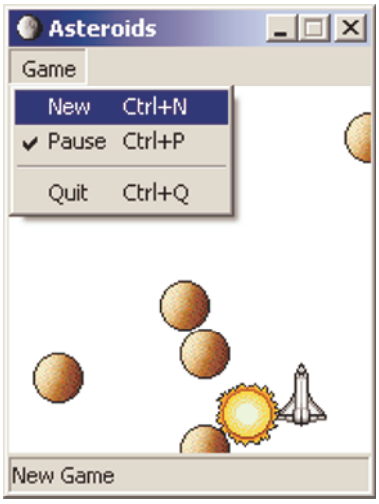

Figure 2. The asteroids game.

First, we retrieve the current rocks and x position of the spaceship. The position of the spaceship, *ship*, is at a fixed y-position. The current rock *positions* are simply the head of the rocks list. The *collisions* list tells for each rock position whether it collides with the ship. Finally, we draw the ship and all the rocks. As a final touch, we also play a sound fragment of an explosion when a collision has happened. The *collide* function just checks if two positions are too close for comfort using standard vector functions from the wxHaskell library:

```
collide pos0 pos1 =
  let distance = vecLength (vecBetween pos0 pos1)
  in distance ⩽ fromIntegral diameter
```

A ship can be drawn using standard drawing primitives, for example, we could draw the ship as a solid red circle:

```
drawShip dc pos =
  circle dc pos (div diameter 2) [brush := brushSolid red]
```

The *circle* function takes a device context, a position, a radius, and a list of properties as arguments. The *brush* attribute determines how the circle is filled. wxHaskell comes with an extensive array of drawing primitives, for example polygons, rounded rectangles, and elliptic arcs. But for a spaceship, it is nicer of course to use bitmaps instead:

```
drawShip dc pos =
  drawBitmap dc ship pos True []
drawRock dc (pos, collides) =
  let picture = if collides then burning else rock
  in drawBitmap dc picture pos True []
```

The *drawBitmap* function takes a device context, a bitmap, a position, the transparency mode, and a list of properties as arguments. The bitmap for a rock is changed to a *burning* ball when it collides with the spaceship. To finish the program, we define the resources that we used:

```
rock    = bitmap "rock.ico"
burning = bitmap "burning.ico"
```

```
ship    = bitmap "ship.ico"
explode = sound "explode.wav"
```

And that is all we need – asteroids in 55 lines of code.

3.1 Extensions

Extending the game with new features is straightforward. For example, to change the speed of the spaceship by pressing the plus or minus key, we just add more event handlers to the frame *f*:

```
on (charKey '-') := set t [interval :~ \i -> i*2]
on (charKey '+') := set t [interval :~ \i -> max 10 (div i 2)]
```

The minus key increments the timer interval, while the plus key decrements it, effectively making the game run slower or faster. The screenshot in Figure 2 also shows a menu and status bar. Here is the code for creating the menu pane:

```
game  <- menuPane       [text  := "&Game"]
new   <- menuItem game  [text  := "&New\tCtrl+N"
                        ,help  := "New game"]
pause <- menuItem game  [text  := "&Pause\tCtrl+P"
                        ,help  := "Pause game"
                        ,checkable := True]
menuLine game
quit  <- menuQuit game  [help  := "Quit the game"]
```

The "&" notation in menu texts signifies the hotkey for that item when the menu has the focus. Behind a tab character we can also specify a menu shortcut key. There is also a structured interface to such accelerator keys, but specifying those keys as part of the menu text proves very convenient in practice. Note that the *pause* menu is a checkable menu item. For the *quit* menu, we use the special *menuQuit* function instead of *menuItem*, as this item is sometimes handled specially on certain platforms, in particular on Mac OS X.

To each new menu item, we attach an appropiate event handler:

```
set new   [on command := asteroids]
set pause [on command := set t [enabled :~ not]]
set quit  [on command := close f]
```

The *quit* menu simply closes the frame. The *pause* menu toggles the enabled state of the timer by applying the *not* function. Turning off the timer effectively pauses the game.[2] The *new* menu is interesting as it starts a completely new asteroids game in another frame. As we don't use any global variables, the new game functions completely independent from any other asteroids game. Finally, we show the menu by specifying the menu bar of the frame:

```
set f [menubar := [game]]
```

Our final extension is a status bar. A status bar consists of status fields that contain text or bitmaps. For our game, a single status field suffices.

```
status <- statusField [text := "Welcome to asteroids"]
set f [statusbar := [status]]
```

[2] Although one can cheat now by changing the x position of the ship while in pause mode.

The *status* is passed to the *advance* function, which updates the status field with the count of rocks that are currently visible:

```
advance status vrocks f =
  do (r:rs) <- get vrocks value
     set vrocks [value := rs]
     set status [text  := "rocks: " ++ show (length r)]
     repaint f
```

4 Design

In the previous section, we have seen how graphical user interfaces in wxHaskell are defined using the imperative *IO* monad. Despite the use of this monad, the examples have a declarative flavour and are much more concise than their imperative counterparts in C++. We believe that the ability to treat *IO* computations as first class values allows us to reach this high level of abstraction: using the ability to defer, modify and combine computations, we can for example use attribute lists to set properties of widgets.

The use of mutable variables to communicate across event handlers is very imperative, though. There has been much research into avoiding mutable state and providing a declarative model for GUI programming. We discuss many of these approaches in the related work section. However, this is still an active research area and we felt it was better to provide a standard monadic interface first. As shown in [13], it is relatively easy to implement a declarative interface on top of a standard monadic interface, and others have already started working on a Fruit [14] interface on top of wxHaskell [35].

4.1 Safety

The wxHaskell library imposes a strong typing discipline on the wxWidgets library. This means that the type checker will reject programs with illegal operations on widgets. Also, the memory management is fully automatic, with the provision that programmers are able to manually manage certain external resources like font descriptors or large bitmaps. The library also checks for *NULL* pointers, raising a Haskell exception instead of triggering a segmentation fault.

Common to many other GUI libraries, wxHaskell still suffers from the *hierarchy problem*: the library imposes a strict hierarchical relation on the created widgets. For example, the program in Figure 1 shows how the buttons and the label all take the parent frame *f* as their first argument. It would be more natural to just create buttons and labels:

```
f   <- frame  [text := "Example"]
lab <- label  [text := "Hello wxHaskell"]
ok  <- button [text := "Ok"]
can <- button [text := "Cancel"]
```

The layout now determines a relation between widgets. We believe that the hierarchical relation between widgets is mostly an artifact of libraries where memory management is explicit: by imposing a strict hierarchical order, a container can automatically discard its child widgets.

Even with the parent argument removed, there are still many ways to make errors in the layout specification. Worse, these errors are

not caught by the type checker but occur at runtime. There are three kind of errors: 'forgetting' widgets, duplication of widgets, and violating the hierarchical order. Here are examples of the last two error kinds.

```
set f  [layout := row 5 [widget ok, widget ok]]   -- duplication
set ok [layout := widget can]                     -- order
```

A potential solution to the *hierarchy problem* is the use of a *linear type* system [7, 45] to express the appropiate constraints. Another solution is to let the layout specification construct the components. One can implement a set of layout combinators that return a nested cartesian product of widget identifiers. The nested cartesian product is used to represent a heterogenous list of identifiers, and combinators that generate those can be implemented along the lines of Baars *et al* [6]. Here is a concrete example of this approach:

```
do (f,(lab,(ok,(can,())))) ← frame (above label
                                    (beside button button))
```

The returned identifiers can now be used to set various properties of all widgets. Using *fixIO* and the recursive *mdo* notation of Erkök and Launchbury [17], we can even arrange things so that widgets can refer to each other at creation time.

We have not adopted this solution for wxHaskell though. First, the syntax of the nested cartesian product is inconvenient for widgets with many components. Furthermore, the order of the identifiers is directly determined by layout; it is very easy to make a small mistake and get a type error in another part of the program. Due to type constraints, the layout combinators can no longer use convenient list syntax to present rows and columns, but fixed arity combinators have to be used. Further research is needed to solve these problems, and maybe record calculi or syntax macros may provide solutions. For now, we feel that the slight chance of invalid layout is acceptable with the given alternatives.

5 Inheritance

Since wxHaskell is based on an object-oriented framework, we need to model the inheritance relationship between different widgets. This relation is encoded using *phantom* types [27, 26]. In essence, wxHaskell widgets are just foreign pointers to C++ objects. For convenience, we use a type synonym to distinguish these object pointers from other pointers:

```
type Object a = Ptr a
```

The type argument *a* is a phantom type: no value of this type is ever present as pointers are just plain machine adresses. The phantom type *a* is only used to encode the inheritance relation of the objects in Haskell. For each C++ class we have a corresponding *phantom data type* to represent this class, for example:

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

We call this a phantom data type as the type is only used in phantom type arguments. As no values of phantom types are ever created, no constructor definition is needed. Currently, only GHC supports phantom data type declarations, and in the library we just supply dummy constructor definitons. Next, we define type synonyms that encode the full inheritance path of a certain class:

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

Using these types, we can impose a strong type discipline on the different kinds of widgets, making it impossible to perform illegal operations on the object pointers. For example, here are the types for the widget creation functions of Figure 1:

```
frame ::                [Prop (Frame ())] → IO (Frame ())
button :: Window a →    [Prop (Button ())] → IO (Button ())
label  :: Window a →    [Prop (Label ())]  → IO (Label ())
```

For now, we can ignore the type of the property lists which are described in more detail in the Section 6. We see how each function creates an object of the appropiate type. A type *C* () denotes an object of exactly class *C*; a type *C a* denotes an object that is at least an instance of class *C*. In the creation functions, the co(ntra) variance is encoded nicely in these types: the function *button* creates an object of exactly class *Button*, but it can be placed in any object that is an instance of the *Window* class. For example:

```
do f ← frame []
   b ← button f []
```

The frame *f* has type *Frame* (). We can use *f* as an argument to *button* since a *Frame* () is an instance of *Window a* – just by expanding the type synonyms we have:

```
Frame () = Window (CFrame ()) ≅ Window a
```

The encoding of (single interface) inheritance using polymorphism and phantom types is simple and effective. Furthermore, type errors from the compiler are usually quite good – especially in comparison with an encoding using Haskell type classes.

6 Attributes and properties

In this section we discuss how we type and implement the *attributes* of widgets. Attributes first appeared in Haskell/DB [27] in the context of databases but proved useful for GUI's too. In Figure 1 we see some examples of widget attributes, like *text* and *layout*. The type of an attribute reflects both the type of the object it belongs to, and the type of the values it can hold. An attribute of type *Attr w a* applies to objects of type *w* that can hold values of type *a*. For example, the *text* attribute for buttons has type:

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

The current value of an attribute can be retrieved using *get*:

```
get :: w → Attr w a → IO a
```

The type of *get* reflects the simple use of polymorphism to connect the type of an attribute to both the widgets it applies to (*w*), and the type of the result (*a*).

Using the $(:=)$ operator, we can combine a value with an attribute. The combination of an attribute with a value is called a *property*. Properties first appeared in Koen Claessen's (unreleased) Yahu library [12], and prove very convenient in practice. In wxHaskell, we use a refined version of the Yahu combinators. Since the value is given, the type of properties is only associated with the type of objects it belongs to. This allows us to combine properties of a certain object into a single homogenous list.

```
(:=) :: Attr w a → a → Prop w
```

Finally, the *set* function assigns a list of properties to an object:

```
set :: w → [Prop w] → IO ()
```

As properties still carry their object parameter, polymorphism ensures that only properties belonging to an object of type *w* can be used. Here is a short example that attaches an exclamation mark to the text label of a button:

```
exclamation :: Button a → IO ()
exclamation b =
  do s ← get b text
     set b [text := s ++ "!"]
```

The update of an attribute is a common operation. The update operator $(:\sim)$ applies a function to an attribute value:

```
(:~) :: Attr w a → (a → a) → Prop w
```

Using this operator in combination with the Haskell section syntax, we can write the previous example as a single concise expression:

```
exclamation b = set b [text :~ (++"!")]
```

6.1 Shared attributes

Many attributes are shared among different objects. For example, in Figure 1, the *text* attribute is used for frames, buttons, and labels. Since the wxWidgets *Window* class provides for a *text* attribute, we could use inheritance to define the *text* attribute for any kind of window:

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

However, this is not such a good definition for a library, as user defined widgets can no longer support this attribute. In wxHaskell, the *text* attribute is therefore defined in a type class, together with an instance for windows:

```
class Textual w where
  text :: Attr w String
instance Textual (Window a) where
  text = ...
```

Here, we mix object inheritance with *ad hoc* overloading: any object that derives from the *Window* class, like buttons and labels, are also an instance of the *Textual* class and support the *text* attribute. This is also very convenient from an implementation perspective – we can implement the *text* attribute in terms of wxWidgets primitives in a single location. If the inheritance was not encoded in the type parameter, we would have to define the *text* attribute for every widget kind separately, i.e. an instance for buttons, another instance for labels, etc. Given that a realistic GUI library like wxWidgets supports at least fifty separate widget kinds, this would quickly become a burden.

The price of this convenience is that we do not adhere to the Haskell98 standard (in the *WX* library). When we expand the type synonym of *Window a*, we get the following instance declaration:

```
instance Textual (Ptr (CObject (CWindow a)))
```

This instance declaration is illegal in Haskell 98 since an instance type must be of the form $(T\ a_1 \ldots a_n)$. This restriction is imposed to prevent someone from defining an overlapping instance, for example:

```
instance Textual (Ptr a)
```

In a sense, the Haskell98 restriction on instance types is too strict: the first instance declaration is safe and unambiguous. Only new instances that possibly overlap with this instance should be rejected. The GHC compiler lifts this restriction and we use the freedom to good effect in the *WX* library.

6.2 Implementation of attributes

Internally, the attribute data type stores the primitive set and get functions. Note that this single definition shows that polymorphism, higher-order functions, and first class computations are very convenient for proper abstraction.

```
data Attr w a = Attr (w → IO a) (w → a → IO ())
```

As an example, we give the full definition of the *text* attribute that uses the primitive *windowGetLabel* and *windowSetLabel* functions of the *WXCore* library:

```
instance Textual (Window a) where
  text = Attr windowGetLabel windowSetLabel
```

The *get* function has a trivial implementation that just extracts the corresponding function from the attribute and applies it:

```
get :: w → Attr w a → IO a
get w (Attr getter setter) = getter w
```

The attentive reader will have noticed already that the assignment operators, $(:=)$ and $(:\sim)$, are really constructors since they start with a colon. In particular, they are the constructors of the property data type:

```
data Prop w = ∀a. (Attr w a) := a
            | ∀a. (Attr w a) :~ (a → a)
```

We use local quantification [24] to hide the value type *a*, which allows us to put properties in homogenous lists. This is again an extension to Haskell98, but it is supported by all major Haskell compilers. The *set* function opens the existentially quantified type through pattern matching:

```
set :: w → [Prop w] → IO ()
set w props
  = mapM_ setone props
  where
    setone (Attr getter setter := x) = setter w x
    setone (Attr getter setter :~ f) = do x ← getter w
                                          setter w (f x)
```

It is well known that an explicit representation of function application requires an existential type. We could have avoided the use of existential types, by defining the assignment and update operators directly as functions. Here is a possible implementation:

```
type Prop w = w → IO ()
(:=) :: Attr w a → a → Prop w
(:=) (Attr getter setter) x = \w → setter w x
set :: w → [Prop w] → IO ()
set w props = mapM_ (\f → f w) props
```

This is the approach taken by the Yahu library. However, this does not allow reflection over the property list, which is used in wxHaskell to implement *creation* attributes (which are beyond the scope of this article).

7 Layout

This section discusses the design of the layout combinators of wxHaskell. The visual layout of widgets inside a parent frame is specified with the *layout* attribute that holds values of the abstract data type *Layout*. Here are some primitive layouts:

```
caption :: String → Layout
space   :: Int → Int → Layout
rule    :: Int → Int → Layout
boxed   :: String → Layout → Layout
```

The *caption* layout creates a static text label, *space* creates an empty area of a certain width and height, and *rule* creates a black area. The *boxed* layout container adds a labeled border around a layout.

Using the *widget* combinator, we can layout any created widget that derives from the *Window* class.The *container* combinator is used for widgets that contain other widgets, like scrolled windows or panels:

```
widget    :: Window a → Layout
container :: Window a → Layout → Layout
```

To allow for user defined widgets, the *widget* combinator is actually part of the *Widget* class, where *Window a* is an instance of *Widget*.

Basic layouts can be combined using the powerful *grid* combinator:

```
grid :: Int → Int → [[Layout]] → Layout
```

The first two arguments determine the amount of space that should be added between the columns and rows of the grid. The last argument is a list of rows, where each row is a list of layouts. The *grid* combinator will lay these elements out as a table where all columns and rows are aligned.

We can already define useful abstractions with these primitives:

```
empty      = space 0 0
hrule w    = rule w 1
vrule h    = rule 1 h
row w xs   = grid w 0 [xs]
column h xs = grid 0 h [[x] | x ← xs]
```

Here is an example of a layout that displays two text entries for retrieving an x- and y-coordinate. The *grid* combinator is used to align the labels and text entries, with 5 pixels between the components.

```
grid 5 5 [[caption "x:", widget xinput]
         ,[caption "y:", widget yinput]]
```

7.1 Alignment, expansion, and stretch

We can see that with the current set of primitive combinators, we can always calculate the *minimum size* of a layout. However, the area in which a layout is displayed can be larger than its minimum size, due to alignment constraints imposed by a *grid*, or due to user interaction when the display area is resized. How a layout is displayed in a larger area is determined by three attributes of a layout: the *alignment*, the *expansion*, and the *stretch*.

The *alignment* of a layout determines where a layout is positioned in the display area. The alignment consists of horizontal and vertical aligment. Ideally, each component can be specified continuously between the edges, but unfortunately, the wxWidgets library only allows us to align centered or towards the edges. There are thus six primitives to specify the alignment of a layout:

```
halignLeft   :: Layout → Layout   -- default
halignRight  :: Layout → Layout
halignCenter :: Layout → Layout

valignTop    :: Layout → Layout   -- default
valignBottom :: Layout → Layout
valignCenter :: Layout → Layout
```

The *expansion* of a layout determines how a layout expands into the display area. There are four possible expansions. By default, a layout is *rigid*, meaning that it won't resize itself to fit the display area. A layout is *shaped* when it will proportionately expand to fill the display area, i.e. it maintaints its aspect ratio. For a *shaped* layout, the alignment is only visible in one direction, depending on the display area.

The other two modes are *hexpand* and *vexpand*, where a layout expands only horizontally or vertically to fit the display area. Again, wxWidgets does not allow the last two modes separately, and we only provide an *expand* combinator that expands in both directions. For such layout, alignment is ignored completely.

```
rigid  :: Layout → Layout   -- default
shaped :: Layout → Layout
expand :: Layout → Layout
```

The *stretch* of a layout determines if a layout demands a larger display area in the horizontal or vertical direction. The previous two attributes, *alignment* and *expansion*, determine how a layout is rendered when the display area is larger than the minimum. In contrast,

the *stretch* determines whether the layout actually gets a larger display area assigned in the first place! By giving a layout stretch, it is assigned all extra space left in the parents' display area.

```
static  :: Layout → Layout   -- default
hstretch :: Layout → Layout
vstretch :: Layout → Layout
stretch  = hstretch ∘ vstretch
```

As a rule, *stretch* is automatically applied to the top layout, which ensures that this layout gets at least all available space assigned to it. For example, the following layout centers an *ok* button horizontally in a frame *f*:

```
set f [layout := halignCenter (widget ok)]
```

Due to the implicit *stretch* this example works as it stands. If this stretch had not been applied, the layout would only be assigned its minimim size as its diplay area, and the centered alignment would have no visible effect. So stretch is not very useful for layouts consisting of a single widget; it only becomes useful in combination with grids.

7.2 Stretch and expansion for grids

Layout containers like *boxed* and *container* automatically inherit the stretch and expansion mode of their children. Furthermore, a *grid* has a special set of rules that determines the stretch of its rows and columns. A column of a grid is horizontally stretchable when all elements of that columnn have horizontal stretch. Dually, a row is vertically stretchable when all elements of that row have vertical stretch. Furthermore, when any row or column is stretchable, the grid will stretch in that direction too and the grid will *expand* to fill assigned area.

This still leaves the question of how extra space is divided amongst stretchable rows and columns. The *weight* attribute is used to proportionally divide space amongst rows and columns. A layout can have a horizontal and vertical (positive) weight:

```
hweight :: Int → Layout → Layout
vweight :: Int → Layout → Layout
```

The default weight of a layout is one. The weight of a row or column is the maximum weight of its elements. The weight of the rows and columns is not propagated to the grid layout itself, which has its own weight.

There are two rules for dividing space amongst rows and columns: first, if all weights of stretchable elements are equal, the space is divided equally amongst those elements. If the weights are differing, the space is divided proportionally according to the weight of the element – i.e. a layout with weight two gets twice as much space as a layout with weight one. The first rule is useful for attaching zero weights to elements, that will cancel out as soon as another element becomes stretchable (with a weight larger than zero). Alas, the current wxWidgets implementation does not provide proportional stretching yet, and wxHaskell disregards all weight attributes at the moment of writing.

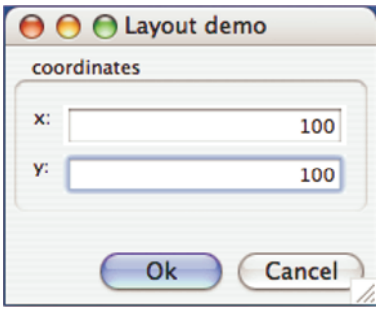

Figure 3. Layout on MacOS X.

7.3 Common layout transformers

With the given set of primitive combinators, we can construct a set of combinators that capture common layout patterns. For example, alignment in the horizontal and vertical direction can be combined:

```
alignCenter       = halignCenter ∘ valignCenter
alignBottomRight  = halignRight  ∘ valignBottom
```

By combining stretch with alignment, we can float a layout in its display area:

```
floatCenter       = stretch ∘ alignCenter
floatBottomRight  = stretch ∘ alignBottomRight
```

Dually, by combining stretch and expansion, layouts will fill the assigned display area:

```
hfill = hstretch ∘ expand
vfill = vstretch ∘ expand
fill  = hfill ∘ vfill
```

Using stretchable empty space, we can emulate much of the behaviour of TEX boxes, as stretchable empty space can be imagined as glue between layouts.

```
hglue = hstretch empty
vglue = vstretch empty
glue  = stretch  empty
```

Using the *glue* combinators in combination with *weight*, it is possible to define the 'primitive' alignment combinators in terms of glue. For example:

```
halignCenter l = row 0 [hweight 0 hglue, l, hweight 0 hglue]
```

Note that we set the horizontal weight of the *hglue* to zero. When the layout *l* stretches horizontally and expands, the entire display area should be assigned to the *l* in order to expand over all the available space. Since the default weight of *l* is one, a proportional division of the available space indeed assigns everything to *l*, mimicking the behaviour of its primitive counterpart.

7.4 Example

Here is a complete example that demonstrates a complicated layout and the convenience of the grid propagation rules. We layout a

frame that displays a form for entering an x and y coordinate, as shown in Figure 3.

```
layoutDemo
  = do f   ← frame    [text := "Layout demo"]
       p   ← panel  f []
       x   ← entry  p [text := "100"]
       y   ← entry  p [text := "100"]
       ok  ← button p [text := "Ok"]
       can ← button p [text := "Cancel"]
       set f [layout :=
         container p $ margin 5 $
         column 5 [boxed "coordinates" $
                     grid 5 5 [[caption "x:", hfill (widget x)]
                               ,[caption "y:", hfill (widget y)]]
                   ,floatBottomRight $
                     row 5 [widget ok, widget can]
                   ]]
```

The *panel* creates an empty widget that manages keyboard navigation control for child widgets[3]. When this frame is resized, the text entries fill the available space horizontally (due to *hfill*), while the ok and cancel buttons float to the bottom right. Due to the propagation rules, the *grid* stretches horizontally and expands, just like the *boxed* layout. Furthermore, the *column* stretches in both directions and expands, and thus the entire layout is resizeable. When the *floatBottomRight* is replaced by an *alignBottomRight*, there is no stretch anymore, and the horizontal stretch of the *boxed* layout is not propagated. In this case, the top layout is no longer resizeable.

We can express the same layout using a TEX approach with *glue*:

```
container p $ margin 5 $
column 0 [boxed "coordinates" $
            grid 5 5 [[caption "x:", hfill (widget x)]
                      ,[caption "y:", hfill (widget y)]]
          ,stretch (vspace 5)
          ,row 0 [hglue, widget ok, hspace 5, widget can]
          ]]
```

Note that we need to be more explicit about the space between elements in a row and column.

7.5 Implementing layout

The implementation of layout combinator library is interesting in the sense that the techniques are generally applicable for declarative abstractions over imperative interfaces [27, 26]. In the case of the wxWidgets library, the imperative interface consists of creating *Sizer* objects that encode the layout constraints imposed by the wxHaskell layout combinators.

Instead of directly creating *Sizer* objects, we first generate an intermediate data structure that represents a canonical encoding of the layout. Besides leading to clearer code, it also enables analysis and transformation of the resulting data structure. For example, we can implement the propagation rules as a separate transformation. Only when the layout is assigned, the data structure is translated into an *IO* value that creates proper *Sizer* objects that implement the layout.

[3] wxHaskell *panel*'s have nothing to do with Java panels that are used for layout.

The *Layout* data type contains a constructor for each primitive layout. Each constructor contains all information to render the layout:

```
data Layout
  = Grid   {attrs :: Attrs, gap :: Size, rows :: [[Layout]]}
  | Widget {attrs :: Attrs, win :: Window ()}
  | Space  {attrs :: Attrs, area :: Size}
  | Label  {attrs :: Attrs, txt  :: String}
  ...
```

All primitive layouts contain an *attrs* field that contains all common layout attributes, like alignment and stretch:

```
data Attrs      = Attrs{stretch_h  :: Bool
                       , stretch_v  :: Bool
                       , align_h    :: Align_h
                       , align_v    :: Align_v
                       , expansion  :: Expansion
                       ...}
data Expansion  = Rigid | Shaped | Expand
data Align_h    = AlignLeft | AlignRight | AlignCenter_h
data Align_v    = AlignTop | AlignBottom | AlignCenter_v
```

The implementation of the basic layout combinators is straightforward:

```
space w h = Space   defaultAttrs (size w h)
widget w  = Widget defaultAttrs (downcastWindow w)
...
```

The implementation of the layout transformers is straightforward too, but somewhat cumbersome due to the lack of syntax for record updates:

```
rigid l    = l{attrs = (attrs l){expansion = Rigid}}
hstretch l = l{attrs = (attrs l){stretch_h  = True}}
...
```

The *grid* combinator is more interesting as we have to apply the propagation rules for stretch and expansion. These rules have to be applied immediately to the attributes to implement the layout transformers faithfully. A separate pass algorithm is also possible, but that would require a more elaborate *Layout* data type with an explicit representation of layout transformers.

```
grid w h rows = Grid gridAttrs (size w h) rows
  where
    gridAttrs = defaultAttrs{
      stretch_v  = any (all (stretch_v ∘ attrs)) rows,
      stretch_h  = any (all (stretch_h ∘ attrs)) (transpose rows),
      expansion = if (stretch_v gridAttrs ∨ stretch_h gridAttrs)
                  then Expand else Static}
```

We can elegantly view rows as columns using *transpose*. Note also that the use of laziness in the definition of *expansion* is not essential.

Now that we made the layout explicit in the *Layout* data structure, we can write a function that interprets a *Layout* structure and generates the appropiate wxWidgets' *Sizer* objects:

```
sizerFromLayout :: Window a → Layout → IO (Sizer ())
```

We will not discuss this function in detail as the interface to *Sizer* objects is beyond the scope of this article. However, with an explicit representation of layout, it is fairly straightforward to create the corresponding *Sizer* objects. The ability to freely combine and manipulate *IO* values as first-class entities during the interpretation of the *Layout* description proves very useful here.

8 Communication with C++

Even though Haskell has an extensive foreign function interface [11], it was still a significant challenge to create the Haskell binding to the wxWidgets C++ library. This section describes the technical difficulties and solutions.

8.1 The calling convention

No current Haskell compiler supports the C++ calling convention, and we do not expect that this situation will change in the near future. The solution adapted by wxHaskell is to expose every C++ function as a C function. Here is an example of a wrapper for the *SetLabel* method of the *Window* class:

```
extern "C"
void wxWindowSetLabel( wxWindow* self, const char* text) {
  self → SetLabel(text);
}
```

We also create a C header file that contains the signature of our wrapper function:

```
extern void wxWindowSetLabel( wxWindow*, const char*);
```

The *wxWindowSetLabel* function has the C calling convention and can readily be called from Haskell using the foreign function interface. We also add some minimal marshalling to make the function callable using Haskell types instead of C types.

```
windowSetLabel :: Window a → String → IO ()
windowSetLabel self text =
  whenValid self (withCString text (wxWindowSetLabel self))
foreign import ccall "wxWindowSetLabel"
  :: Ptr a → Ptr CChar → IO ()
```

To avoid accidental mistakes in the foreign definition, we include the C header file when compiling this Haskell module. GHC can be instructed to include a C header file using the *−#include* flag.

Unfortunately, this is not the entire story. The C++ library is linked with the C++ runtime library, while the Haskell program is linked using the C runtime library – resulting in link errors on most platforms. wxHaskell avoids these problems by compiling the C++ code into a dynamic link library. A drawback of this approach is that the linker can not perform dead code elimination and the entire wxWidgets library is included in the resulting dynamic link library. Of course, it can also save space, as this library is shared among all wxHaskell applications.

8.2 wxDirect

If there were only few functions in the wxWidgets library, we could write these C wrappers by hand. However, wxWidgets contains more than 500 classes with about 4000 methods. Ideally, we would have a special tool that could read C++ header files and generate the needed wrappers for us. The SWIG toolkit [8] tries to do exactly this, but writing a SWIG binding for Haskell and the corresponding binding specification for wxWidgets is still a lot of work. Another option is to use a tool like SWIG to generate IDL from the C++ headers and to use H/Direct [18, 19, 26] to generate the binding.

For wxHaskell, we opted for a more pragmatic solution. The wxEiffel library [39] contains already thousands of hand written C wrappers for wxWidgets together with a header file containing the signatures. wxHaskell uses the same C wrappers for the Haskell binding. The Haskell wrappers and foreign import declarations are generated using a custom tool called wxDirect. This tool uses Parsec [28] to parse the signatures in the C header and generates appropiate Haskell wrappers. As the data types in wxWidgets are limited to basic C types and C++ objects, the marshalling translation much simpler than that of a general tool like H/Direct.

As argued in [18, 19, 26], a plain C signature has not enough information to generate proper marshalling code. Using C macros, we annotated the C signatures with extra information. The signature of *wxWindowSetLabel* is for example:

```
void wxWindowSetLabel( Self(wxWindow) self, String text);
```

Macros like *Self* provide wxDirect with enough information to generate proper marshalling code and corresponding Haskell type signatures. When used by the C compiler, the macros expand to the previous plain C signature. This approach means that changes in the interface of wxWidgets require manual correction of the C wrappers, but fortunately, this interface has been stable for years now.

8.3 Performance

The performance of wxHaskell applications with regard to GUI operations is very good, and wxHaskell applications are generally indistinguishable from “native” applications written with MFC or GTK for example. This is hardly surprising, as all the hard work is done by the underlying C++ library – Haskell is just used as the glue language for the proper wxWidget calls. For the same reason, the memory consumption with regard to GUI operations is also about the same as that of native applications.

One of the largest wxHaskell programs is NetEdit: a Bayesian belief network editor that consists of about 4000 lines of wxHaskell specific code. On Windows XP, NetEdit uses about 12mb of memory for large belief networks of more than 50 nodes. The performance of the drawing routines is so good that NetEdit can use a naïve redraw algorithm without any noticable delays for the user.

The binaries generated with wxHaskell tend to be rather large though – GHC generates a 3mb binary for NetEdit. The use of a compressor like UPX can reduce the size of the executable to about 600kb. The shared library for wxHaskell generated by GCC is also about 3mb. On Windows platforms, we use Visual C++ to generate the shared library which approximately reduces the size to 2mb, which becomes 700kb after UPX compression.

9 Related work

There has been a lot of research on functional GUI libraries. Many of these libraries have a monadic interface. Haggis [20] is build on X Windows and uses concurrency to achieve a high level of composition between widgets. The Gtk2Hs and Gtk+Hs [42] libraries use the Gtk library and, like wxHaskell, provide an extensive array of widgets. Many libraries use the portable Tk framework as their GUI platform. HTk [23] is an extensive library that provides a sophisticated concurrent event model [36]. TkGofer [44, 13] is an elegant library for the Gofer interpreter that pioneered the use of type classes to model inheritance relations. Yahu [12] is an improved (but unreleased) version of TkGofer for Haskell that first used property lists to set attributes. The HToolkit [5] library has similar goals as wxHaskell but implements its own C wrapper around the win32 and Gtk interface.

Besides monadic libraries, there has been a lot of research into more declarative GUI libraries. Functional reactive animations (Fran) [16] elegantly described declarative animations as continuous functions from time to pictures. This idea was used in FranTk [38, 37] to model graphical user interfaces. Functional reactive programming [46] is a development where arrows [22, 32] are used to fix space-time leaks in Fran. The Fruit library [14, 15] uses these techniques in the context of GUI programming. In contrast to Fran, imperative streams [40] use discrete time streams instead of continuous functions to model animation and GUI's.

One of the most well known functional models for GUI programming is Fudgets [9, 10, 33]. The extensive Fudget library uses X windows and is supported by many Haskell compilers. The Fudget combinators give a rigid structure to the data flow in a program, and the Gadgets framework [29] introduces the concept of *wire*s to present a more flexible interface.

Many GUI libraries are implemented for other functional languages. A well known library is the ObjectIO library for Clean [2, 4, 3] that has partly been ported to Haskell [1]. This library uses uniqueness types [7] to safely encapsulate side effects. LablGtk [21] is a binding for O'Caml to the Gtk library and uses a label calculus to model property lists. Exene [34] is a concurrent GUI library for ML that uses X Windows.

H/Direct [18, 19, 26] described phantom types to model single interface inheritance. Another technique to model inheritance, that relies on multiple parameter type classes and functional dependencies, was described by Pang and Chakravarty [31, 30]. Phantom types were discussed as a general technique to impose a strong type discipline on untyped interfaces by Leijen [27, 26].

10 Conclusion

We have learned an important lesson from wxHaskell: do not write your own GUI library! By using the portable and well-maintained wxWidgets C++ library, we were able to create an industrial strength GUI library for Haskell in a relatively short time frame. Furthermore, we have shown how distinctive features of Haskell, like parameteric polymorphism, higher-order functions, and first-class computations, can be used to present a concise and elegant monadic interface to program GUI's. The resulting programs tend to be much shorter, and more concise, than their counterparts in C++.

In the future, we hope to extend the *WX* library with more abstractions and more widgets. Furthermore, we hope that wxHaskell can become a platform for research into more declarative models for programming GUI's.

11 Acknowledgements

wxHaskell could not have existed without the effort of many developers on wxWidgets and wxEiffel, in particular Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn, Uwe Sanders, and many others. Koen Claessen's Yahu library provided the inspiration for property lists in wxHaskell.

12 References

[1] P. Achten and S. Peyton Jones. Porting the Clean object I/O library to Haskell. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages (2000)*, pages 194–213, 2000.

[2] P. Achten and M. Plasmeijer. The beauty and the beast. Technical Report 93–03, Research Inst. for Declarative Systems, Dept. of Informatics, University of Nijmegen, Mar. 1993.

[3] P. Achten and M. J. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[4] P. Achten, J. van Groningen, and M. Plasmeijer. High level specification of I/O in functional languages. In J. Launchbury and P. Sansom, editors, *Workshop Notes in Computer Science*, pages 1–17. Springer-Verlag, 1993. Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 June 1992.

[5] K. A. Angelov. The HToolkit project. `http://htoolkit.sourceforge.net`.

[6] A. Baars, A. Löh, and D. Swierstra. Parsing permutation phrases. In R. Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 171–182. Elsevier, 2001.

[7] E. Barendsen and S. Smetsers. Uniqueness Type Inference. In M. Hermenegildo and S. Swierstra, editors, *7th International Symposium on Programming Language Implementation and Logic Programming (PLILP'95), Utrecht, The Netherlands*, volume 982 of *LNCS*, pages 189–206. Springer-Verlag, 1995.

[8] D. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++ . In *4th annual Tcl/Tk workshop*, Monterey, CA, July 1996.

[9] M. Carlsson and T. Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *Functional Programming and Computer Architectures (FPCA)*, pages 321–330. ACM press, June 1993.

[10] M. Carlsson and T. Hallgren. *Fudgets – Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Gothenburg University, 1998.

[11] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. Peyton-Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 foreign function interface 1.0: an addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi`, Dec. 2003.

[12] K. Claessen. The Yahu library. `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/afp/yahu.html`.

[13] K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *2nd International Conference on Functional programming (ICFP)*, pages 251–262, 1997. Also appeared in ACM SIGPLAN Notices 32, 8, (Aug. 1997).

[14] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *ACM Sigplan 2001 Haskell Workshop*, Sept. 2001.

[15] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM Press, 2003.

[16] C. Elliott and P. Hudak. Functional reactive animation. In *The proceedings of the 1997 ACM Sigplan International Conference on Functional Programming (ICFP97)*, pages 263–273. ACM press, 1997.

[17] L. Erkök and J. Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, Sept. 2000.

[18] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A Binary Foreign Language Interface to Haskell. In *The International Conference on Functional Programming (ICFP)*, Baltimore, USA, 1998. Also appeared in ACM SIGPLAN Notices 34, 1, (Jan. 1999).

[19] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling hell from heaven and heaven from hell. In *The International Conference on Functional Programming (ICFP)*, Paris, France, 1999. Also appeared in ACM SIGPLAN Notices 34, 9, (Sep. 1999).

[20] S. Finne and S. Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, 1995.

[21] J. Garrigue. The LablGtk library. `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`.

[22] J. Hughes. Generalising monads to arrows. In *Science of Computer Programming*, volume 37, pages 67–111, May 2000.

[23] E. Karlsen, G. Russell, A. Lüdtke, and C. Lüth. The HTk library. `http://www.informatik.uni-bremen.de/htk`.

[24] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[25] D. Leijen. The wxHaskell library. `http://wxhaskell.sourceforge.net`.

[26] D. Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, 2003.

[27] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Second USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, Oct. 1999. USENIX Association. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[28] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[29] R. Noble and C. Runciman. Gadgets: Lazy Functional Components for Graphical User Interfaces. In M. Hermenegildo and S. D. Swierstra, editors, *PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 321–340. Springer-Verlag, Sept. 1995.

[30] A. T. H. Pang. Binding Haskell to object-oriented component systems via reflection. Master's thesis, The University of New South Wales, School of Computer Science and Engineering, June 2003. `http://www.algorithm.com.au/files/reflection/reflection.pdf`.

[31] A. T. H. Pang and M. M. T. Chakravarty. Interfacing Haskell with object-oriented languages. In G. Michaelson and P. Trinder, editors, *15th International Workshop on the Implementation of Functional Languages (IFL'03)*, LNCS. Springer-Verlag, 2004.

[32] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.

[33] A. Reid and S. Singh. Implementing fudgets with standard widget sets. In *Glasgow Functional Programming workshop*, pages 222–235. Springer-Verlag, 1993.

[34] J. H. Reppy. *Higher Order Concurrency*. PhD thesis, Cornell University, 1992.

[35] B. Robinson. wxFruit: A practical GUI toolkit for functional reactive programming. `http://zoo.cs.yale.edu/classes/cs490/03-04b/bartholomew.robinson`.

[36] G. Russell. Events in Haskell, and how to implement them. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 157–168, 2001.

[37] M. Sage. The FranTk library. `http://www.haskell.org/FranTk`.

[38] M. Sage. FranTk – a declarative GUI language for Haskell. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 106–117. ACM Press, 2000.

[39] U. Sander et al. The wxEiffel library. `http://wxeiffel.sourceforge.net`.

[40] E. Scholz. Imperative streams - a monadic combinator library for synchronous programming. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 261–272. ACM Press, 1998.

[41] M. Schrage. *Proxima: a generic presentation oriented XML editor*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, 2004.

[42] A. Simons and M. Chakravarty. The Gtk2Hs library. `http://gtk2hs.sourceforge.net`.

[43] J. Smart, R. Roebling, V. Zeitlin, R. Dunn, et al. The wxWidgets library. `http://www.wxwidgets.org`.

[44] T. Vullinghs, D. Tuinman, and W. Schulte. Lightweight GUIs for functional programming. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 341–356. Springer-Verlag, 1995.

[45] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.

[46] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 146–156. ACM Press, 2001.