# Advanced Functional Programming (5)
# wxHaskell

Bastiaan Heeren

Center for Software Technology, Universiteit Utrecht
http://www.cs.uu.nl/groups/ST/

February 19, 2007

# Overview


1. Introduction to wxHaskell
2. Designing a GUI library
   - Inheritance
   - Sharing attributes
   - Improving usability
   - Layout combinators
3. Beyond wxHaskell
4. Agenda

# What is wxHaskell?

- A portable GUI library for Haskell built on top of the wxWidgets library (written in C++)
- Two layers: a “core” binding (WXCore) and a haskellized layer (WX)
- It is huge: 2800 methods, 1400 constants, 500 classes
- Portable, with native look-and-feel (including GTK, Windows, X11, and MacOS X)
- Designed and implemented by Daan Leijen (Microsoft Research, Redmond – before, UU)
- First release: July 2003

# What is wxHaskell?

- A portable GUI library for Haskell built on top of the wxWidgets library (written in C++)
- Two layers: a “core” binding (WXCore) and a haskellized layer (WX)
- It is huge: 2800 methods, 1400 constants, 500 classes
- Portable, with native look-and-feel (including GTK, Windows, X11, and MacOS X)
- Designed and implemented by D
  Research, Redmond – before, UU)
- First release: July 2003

Demo: *timeflows*

# Source code of the *timeflows* demo

## Types and Options

```
-- Time is in seconds, represented as a double.
type Time = Double

-- The (mouse) history consists of time/position pairs (and is never null)
type History = [(Time, Point)]

-- The total time lag of the words behind the mouse cursor
timeSpan :: Time
timeSpan = 1

-- The flowing text
flowText = "Time flows like a river"

-- The font style
flowFont = fontSwiss{_fontSize = 16}
```

## The GUI

```
main = start timeFlows
timeFlows =
  do vmouseHistory ← newIORef [(0, pt 0 0)]   -- never null!

     -- create a frame.
     f ← frame   [text      := flowText]
     p ← panel f [bgcolor := white     ]      -- draw in a panel

     -- set event handlers
     set p       [on paint   := onPaint vmouseHistory
                 ,on idle    := onIdle   vmouseHistory p
                 ,on drag    := onDrag  vmouseHistory
                 ]

     -- set layout
     set f       [layout     := fill (widget p)
                 ,clientSize := sz 300 300      -- initial size
                 ]
```

## Repaint event handler

```
onPaint vmouseHistory dc viewArea
  = do history  ← readIORef vmouseHistory
       time     ← getTime
         -- draw trace line
       polyline dc (map snd history) [penColor := lightgrey]
         -- draw the words
       set dc [font := flowFont]
       mapM_ drawWord (wordPositions history timeSpan time flowText)

  where
    drawWord (pos, word)
      = do    -- center word
              sz ← getTextExtent dc word
              let newX = pointX pos − (sizeW sz `div` 2)
                  newY = pointY pos − (sizeH sz `div` 2)
                -- and draw it.
              drawText dc word (pt newX newY) []
```

## Idle event handler

```
onIdle vmouseHistory win
  = do history ← readIORef vmouseHistory
       if (null (tail history))
         then do    -- don't call idle again until some other event happens
                    return False
         else do time ← getTime
                    repaint win
                      -- prune the history
                    writeIORef vmouseHistory (prune time history)
                    return True

  where
      -- prune the history: only remember time/position pairs up to
      -- a certain time span.
    prune time (h : hs)
      = h : takeWhile (after (time − timeSpan)) hs

    after time (t, p) = time ⩽ t
```

## Mouse drag event handler and helper functions

```
onDrag vmouseHistory mousePos
  = do time ← getTime
       -- prepend a new time/position pair
       modifyIORef vmouseHistory ((time, mousePos):)

  -- Tuple each word in a string with its historic position, given a mouse
  -- history, a time span, and current time.
wordPositions :: History → Time → Time → String → [(Point, String)]
wordPositions history timeSpan time = ...

  -- Get the current Time
getTime :: IO Time
getTime
  = do picoSecs ← getCPUTime
       let time = (fromIntegral picoSecs) / 1.0e12
       return time
```

# The design of a GUI library

Four issues we have to address:

- wxWidgets (C++) relies on inheritance. Can we model this with Haskell's type system?
- Different controls have the same attributes (e.g., text, or certain event handlers)
- Should we introduce special syntax for writing GUI applications?
- How can we specify the visual layout of widgets (inside a parent frame)?

# Inheritance

**Solution:** Use phantom types for modeling inheritance!

In essence, all widgets are just foreign pointers to C++ objects.

$$\textbf{type}\ \mathit{Object}\ a = \mathit{Ptr}\ a$$

# Inheritance

**Solution:** Use phantom types for modeling inheritance!

In essence, all widgets are just foreign pointers to C++ objects.

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

For each C++ class we introduce one phantom data type:

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

Type variable *a* is phantom: no value of this type is present!

# Inheritance (2)

Define type synonyms that encode the full inheritance path (compare with a virtual method table):

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Window}\ a & = \mathit{Object} & (\mathit{CWindow}\ a) \\
\textbf{type}\ \mathit{Frame}\ a & = \mathit{Window} & (\mathit{CFrame}\ a) \\
\textbf{type}\ \mathit{Control}\ a & = \mathit{Window} & (\mathit{CControl}\ a) \\
\textbf{type}\ \mathit{Button}\ a & = \mathit{Control} & (\mathit{CButton}\ a)
\end{array}
$$

Idea:

- $\mathit{Control}\ a$ denotes an object that is *at least* an instance of *Control* (for instance, a button)
- $\mathit{Control}\ ()$ denotes *exactly* a *Control* object

# Inheritance (3)

Does this solve the problem?

Not yet: the functions that operate on these objects should decide which of the two variants to use.

$$\begin{aligned} &\mathit{button} && :: \mathit{Window}\ \ldots \rightarrow [\mathit{Prop}\ (\mathit{Button}\ \ldots)] \rightarrow \mathit{IO}\ (\mathit{Button}\ \ldots) \\ &\mathit{setButtonText} && :: \mathit{Button}\ \ldots \rightarrow \mathit{String} \rightarrow \mathit{IO}\ () \end{aligned}$$

Question: fill in the dots

# Inheritance (3)

Does this solve the problem?

Not yet: the functions that operate on these objects should decide which of the two variants to use.

$$\begin{aligned} &button && :: Window\ \ldots \rightarrow [Prop\ (Button\ \ldots)] \rightarrow IO\ (Button\ \ldots) \\ &setButtonText && :: Button\ \ldots \rightarrow String \rightarrow IO\ () \end{aligned}$$

Question: fill in the dots

$$\begin{aligned} &button && :: Window\ a \rightarrow [Prop\ (Button\ ())] \rightarrow IO\ (Button\ ()) \\ &setButtonText && :: Button\ a \rightarrow String \rightarrow IO\ () \end{aligned}$$

Related to co-/contravariance found in certain type systems

# The design of a GUI library

Four issues we have to address:

- wxWidgets (C++) relies on inheritance. Can we model this with Haskell's type system? **Solved**
- Different controls have the same attributes (e.g., text, or certain event handlers)
- Should we introduce special syntax for writing GUI applications?
- How can we specify the visual layout of widgets (inside a parent frame)?

# Sharing attributes

**Solution:** Use type classes for shared attributes!
Internally, an attribute stores the (primitive) set and get functions.

$$\textbf{data}\ \mathit{Attr}\ w\ a = \mathit{Attr}\ (w \to \mathit{IO}\ a)\ (w \to a \to \mathit{IO}\ ())$$

# Sharing attributes

**Solution:** Use type classes for shared attributes!
Internally, an attribute stores the (primitive) set and get functions.

$$\textbf{data}\ \mathit{Attr}\ w\ a = \mathit{Attr}\ (w \to \mathit{IO}\ a)\ (w \to a \to \mathit{IO}\ ())$$

Introduce a type class for each attribute:

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

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

Likewise, we have *Literate*, *Dimensions*, *Colored*, *Visible*, etcetera.

# Sharing attributes

**Solution:** Use type classes for shared attributes!
Internally, an attribute stores the (primitive) set and get functions.

$$\textbf{data}\ \mathit{Attr}\ w\ a = \mathit{Attr}\ (w \to \mathit{IO}\ a)\ (w \to a \to \mathit{IO}\ ())$$

Introduce a type class for ea

Disadvantage: it is sometimes hard to find what attributes are available: instance definitions might be given on some class higher in the hierarchy.

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

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

Likewise, we have *Literate*, *Dimensions*, *Colored*, *Visible*, etcetera.

# Access to attributes

$$get :: w \rightarrow Attr\ w\ a \rightarrow IO\ a$$
$$set :: w \rightarrow Attr\ w\ a \rightarrow a \rightarrow IO\ a$$

Suppose that $f :: Frame\ ()$, then:

$$set\ f\ text :: String \rightarrow IO\ ()$$
$$get\ f\ text :: IO\ String$$

# Access to attributes

$$get :: w \rightarrow Attr\ w\ a \rightarrow IO\ a$$
$$set :: w \rightarrow Attr\ w\ a \rightarrow a \rightarrow IO\ a$$

Suppose that $f :: Frame\ ()$, then:

$$set\ f\ text :: String \rightarrow IO\ ()$$
$$get\ f\ text :: IO\ String$$

More importantly,

$$set\ (\ldots :: Bitmap\ ())\ text$$

results in a type error:

```
No instance for (Textual (Bitmap ()))
  arising from use of ‘text’ at <No locn>
In the second argument of ‘set’, namely ‘text’
```

# The design of a GUI library

Four issues we have to address:

- wxWidgets (C++) relies on inheritance. Can we model this with Haskell's type system? **Solved**
- Different controls have the same attributes (e.g., text, or certain event handlers) **Solved**
- Should we introduce special syntax for writing GUI applications?
- How can we specify the visual layout of widgets (inside a parent frame)?

# Improving usability

**Solution:** Don't introduce special syntax. Instead, we (ab)use Haskell's syntax and extensions.

A property is a pair consisting of an attribute and a (matching) value. Hence, we could define:

$$\textbf{type}\ \mathit{Prop}\ w\ a = (\mathit{Attr}\ w\ a, a)$$

# Improving usability

**Solution:** Don't introduce special syntax. Instead, we (ab)use Haskell's syntax and extensions.

A property is a pair consisting of an attribute and a (matching) value. Hence, we could define:

$$\textbf{type}\ \mathit{Prop}\ w\ a = (\mathit{Attr}\ w\ a, a)$$

Or even better: use an infix constructor.

$$\textbf{data}\ \mathit{Prop}\ w\ a = (\mathit{Attr}\ w\ a) := a$$

Note that $:=$ is an ordinary data constructor!

# Improving usability (2)

$$\textbf{data}\ \mathit{Prop}\ w\ a = (\mathit{Attr}\ w\ a) := a$$

We can do even better: hide the type argument *a*.

# Improving usability (2)

$$\textbf{data}\ \mathit{Prop}\ w\ a = (\mathit{Attr}\ w\ a) := a$$

We can do even better: hide the type argument *a*.

$$\textbf{data}\ \mathit{Prop}\ w = \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) := a$$

# Improving usability (2)

$$\textbf{data}\ \mathit{Prop}\ w\ a = (\mathit{Attr}\ w\ a) := a$$

We can do even better: hide the type argument $a$.

$$\begin{aligned}\textbf{data}\ \mathit{Prop}\ w = \ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) := a \\ \mid\ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) :\sim (a \to a) \\ \mid\ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) ::= (w \to a) \\ \mid\ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) ::\sim (w \to a \to a)\end{aligned}$$

# Improving usability (2)

$$\textbf{data}\ \mathit{Prop}\ w\ a = (\mathit{Attr}\ w\ a) := a$$

We can do even better: hide the type argument *a*.

$$\begin{aligned}\textbf{data}\ \mathit{Prop}\ w = \ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) := a \\ \mid\ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) :\sim (a \to a) \\ \mid\ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) ::= (w \to a) \\ \mid\ & \textbf{forall}\ a\ .\ (\mathit{Attr}\ w\ a) ::\sim (w \to a \to a)\end{aligned}$$

**The actual types of *get* and *set*:**

$$\mathit{get} :: w \to \mathit{Attr}\ w\ a \to \mathit{IO}\ a$$
$$\mathit{set} :: w \to [\mathit{Prop}\ w] \to \mathit{IO}\ ()$$

# Improving usability (2)

**data** *Prop w a* = (*Attr w a*) := *a*

W

**d**

The (smart) constructor of a widget must be passed a list of properties for initialization.

$$
\begin{array}{lll}
\mathit{frame} & :: & [\mathit{Prop}\ (\mathit{Frame}\ ())] \to \mathit{IO}\ (\mathit{Frame}\ ()) \\
\mathit{button} & :: \mathit{Window}\ a \to & [\mathit{Prop}\ (\mathit{Button}\ ())] \to \mathit{IO}\ (\mathit{Button}\ ()) \\
\mathit{staticText} & :: \mathit{Window}\ a \to & [\mathit{Prop}\ (\mathit{StaticText}\ ())] \to \mathit{IO}\ (\mathit{StaticText}\ ())
\end{array}
$$

**The actual types of *get* and *set*:**

$$\mathit{get} :: w \to \mathit{Attr}\ w\ a \to \mathit{IO}\ a$$
$$\mathit{set} :: w \to [\mathit{Prop}\ w] \to \mathit{IO}\ ()$$

# Event handlers

What about event handlers?

$$
\begin{aligned}
gui = \mathbf{do}\ \ & f \leftarrow frame\ [\,] \\
& p \leftarrow panel\ f\ [on\ paint := myPaintHandler\,] \\
& \ldots
\end{aligned}
$$

# Event handlers

What about event handlers?

```
gui = do  f  ← frame []
          p  ← panel f [on paint := myPaintHandler]
          ...
```

An event is just an attribute (which stores a *callback* function).

```
data Event w a = Event (Attr w a)

on      :: Event w a → Attr w a

paint   :: Paint w     ⇒ Event w (DC () → Rect → IO ())
mouse   :: Reactive w  ⇒ Event w (EventMouse → IO ())
motion  :: Reactive w  ⇒ Event w (Point → IO ())
click   :: Reactive w  ⇒ Event w (Point → IO ())
command :: Commanding w ⇒ Event w (IO ())
```

# The design of a GUI library

Four issues we have to address:

- wxWidgets (C++) relies on inheritance. Can we model this with Haskell's type system? **Solved**
- Different controls have the same attributes (e.g., text, or certain event handlers) **Solved**
- Should we introduce special syntax for writing GUI applications? **Solved**
- How can we specify the visual layout of widgets (inside a parent frame)?

# Layout combinators

Layout is specified with a set of layout combinators:

## Primitive combinators

$$
\begin{aligned}
&\mathit{label} && :: \mathit{String} \rightarrow \mathit{Layout} \\
&\mathit{space} && :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Layout} \\
&\mathit{rule} && :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Layout} \\
&\mathit{boxed} && :: \mathit{String} \rightarrow \mathit{Layout} \rightarrow \mathit{Layout} \\
&\mathit{grid} && :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow [[\mathit{Layout}]] \rightarrow \mathit{Layout} \\
&\mathit{widget} && :: \mathit{Window}\ a \rightarrow \mathit{Layout}
\end{aligned}
$$

# Layout combinators

Layout is specified with a set of layout combinators:

## Primitive combinators

$$
\begin{aligned}
&\mathit{label} && :: \mathit{String} \rightarrow \mathit{Layout} \\
&\mathit{space} && :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Layout} \\
&\mathit{rule} && :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Layout} \\
&\mathit{boxed} && :: \mathit{String} \rightarrow \mathit{Layout} \rightarrow \mathit{Layout} \\
&\mathit{grid} && :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow [[\mathit{Layout}]] \rightarrow \mathit{Layout} \\
&\mathit{widget} && :: \mathit{Window}\ a \rightarrow \mathit{Layout}
\end{aligned}
$$

Dealing with an area larger than its minimal size

- Alignment: positioning, both horizontal and vertical
- Expansion: resize to fill the display area (rigid/shaped/expand)
- Stretch: determines whether the layout gets a larger display area from its parent

Details can be found in the paper.

# Shortcomings

Observations:

- Structuring code for a GUI can be tricky:
  - Stateful information
  - Organizing callback functions
- Most widgets are not typed:
  - The type of a text entry's value is *String*, even if the entered value should be an integer.
  - The selection of a group of radio buttons is of type *Int*

# Shortcomings

Observations:

- Structuring code for a GUI can be tricky:
  - Stateful information
  - Organizing callback functions
- Most widgets are not typed:
  - The type of a text entry's value is *String*, even if the entered value should be an integer.
  - The selection of a group of radio buttons is of type *Int*

We will use:

- The model/view/controller design pattern from the OO community (but how to do this in Haskell?)
- Extended typed controls (XTC) – a typed layer on top of existing wxHaskell controls

# Extended typed controls

The idea:

```
class TypedValue w t | w → t where
  typedValue :: Attr w t

instance TypedValue (TypedRadioBox t ()) t where ...
```

# Extended typed controls

The idea:

```
class TypedValue w t | w → t where
  typedValue :: Attr w t

instance TypedValue (TypedRadioBox t ()) t where ...
```

```
class Observable obs val | obs → val where
  getValue      :: obs → IO val
  setValue      :: obs → val → IO ()
  getObservers  :: obs → IO [val → IO ()]
  addObserver   :: obs → (val → IO ()) → IO ()

instance Observable (TypedRadioBox t ()) t where ...
```

# Extended typed controls

The idea:

```
class TypedValue w t | w → t where
  typedValue :: Attr w t

instance TypedValue (TypedRadioBox t ()) t where ...
```

```
class Observable obs val | obs → val where
  getValue      :: obs → IO val
  setValue      :: obs → val → IO ()
  getObservers  :: obs → IO [val → IO ()]
  addObserver   :: obs → (val → IO ()) → IO ()

instance Observable (TypedRadioBox t ()) t where ...
```

- Topic of the second exercise
- Demo: a simple CD shop

# Agenda

1. Read *Chapter 2: H/Direct: a binary language interface for Haskell* from Daan's PhD thesis.
2. Study Sections 6–8 of *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell (Simon Peyton Jones)*.
3. Friday: Deadline Exercise 1 (state monad)
4. Next week: each ant team makes an appointment with me to discuss their approach and progress so far.

## Next lecture:

February 21 (Wednesday): Foreign Function Interface and an introduction to Functional Data Structures.