Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

C12. Grafical User Interfaces: wxHaskell

Doaitse Swierstra

Utrecht University

December 3, 2012

Graphical User Interfaces

Reading and writing to a terminal window and/or files is not so interesting for most applications; instead we want to progra graphical user interfaces.

The ideal GUI-library

Requirements for an ideal GUI-library:

- Efficiënt
- Portable
- Native look-and-feel
- A lot of standard functionality
- Easy to use

In case of Haskell:

- Possibility to abstrcat
- Fully typed, guarding against wrong usage of the libary

Implementatie

In principle we can build up a GUI-library from the ground. THis is however a temendous amount of work.

Better idea: make use of existing infra structure.

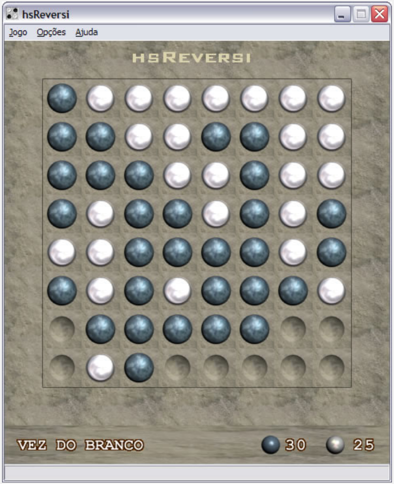
wxHaskell

wxHaskell

(Daan Leijen, Utrecht, Haskell Workshop 2004)

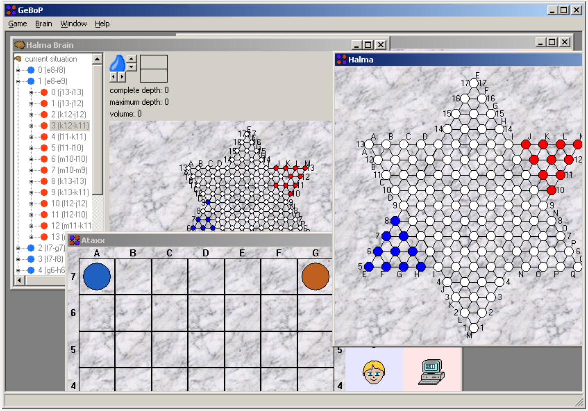
- 'Portable and concise'
- Constructed on top of wxWidgets
- Free (open source)
- Well documented
- `http://haskell.org/haskellwiki/WxHaskell/`

hsReversi



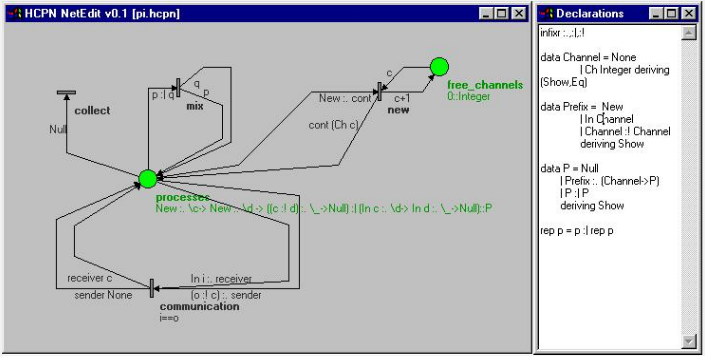
(Lucas Torreão, Emanoel Barreiros, Hilda Borborema en Keldjan Alves)

GeBoP



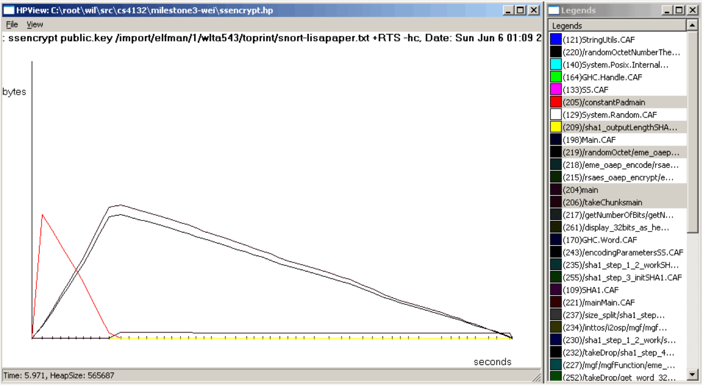
(Maarten Löffler)

HCPN



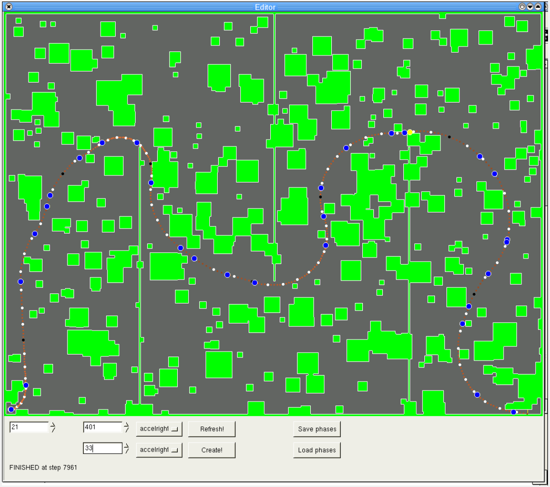
(Claus Reinke)

HPView



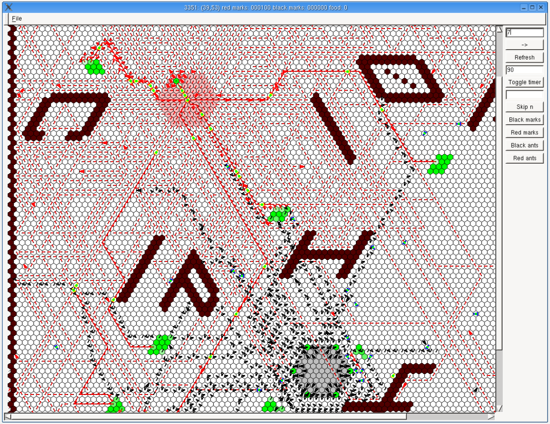
(Wei Tan)

Track editor



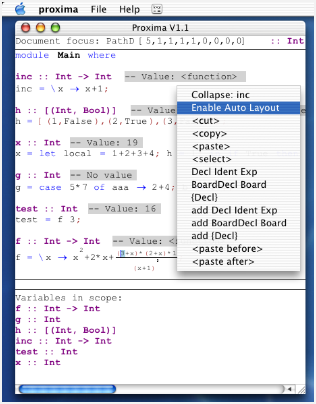
(Ade Azurat, Arthur Baars, Eelco Dolstra en Andres Löh)

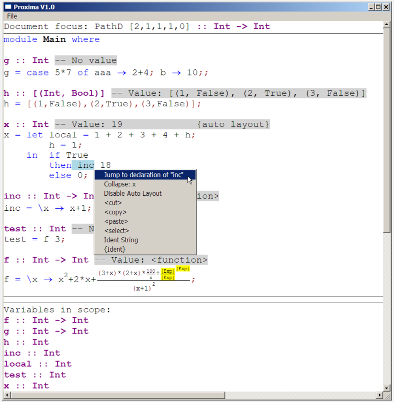
Ant simulator



(Duncan Coutts, Andres Löh, Ian Lynagh
en Ganesh Sittampalam)

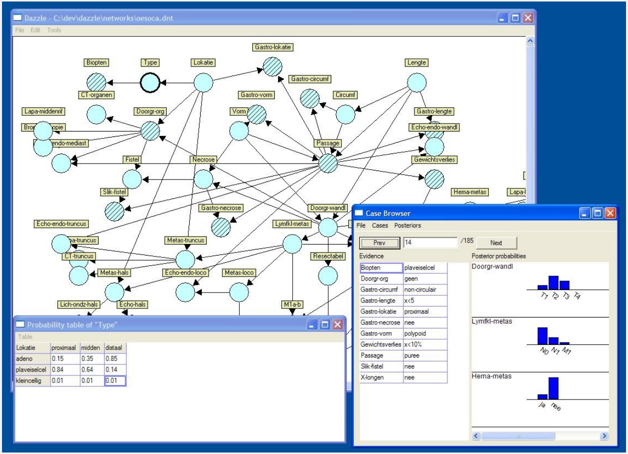
Proxima





(Martijn Schrage)

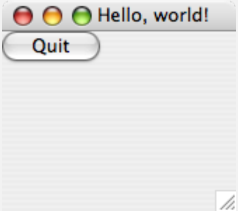
Dazzle



(Martijn Schrage en Arjan van IJzendoorn)

Hello, world!

Our first wxHaskell-program:



- A friendly greeting the the title bar
- A button to close the window

Hello, world!

```
import Graphics.UI.WX

main :: IO ()
main = start hello

hello :: IO ()
hello = do f    ← frame    [text := "Hello, world!"]
           quit ← button f [text := "Quit"]
           set quit [on command := close f]
           set f    [layout     := widget quit]
```

Hello, world!

From source code to executable (after installing the *wx* package):

ghc Hello.hs

```
[1 of 1] Compiling Main (Hello.hs, Hello.o)
Linking Hello ...
```

Interface construction

Initialisation of a wxHaskell-program:

$$start :: IO\ () \rightarrow IO\ ()$$

Functions which create an element in the interface get as argument a possible parent in the widget tree and a *list of properties*:

$$\begin{aligned}
frame &:: [Prop\ (Frame\ ())] \rightarrow IO\ (Frame\ ()) \\
button &:: Window\ a \rightarrow [Prop\ (Button\ ())] \rightarrow IO\ (Button\ ()) \\
panel &:: Window\ a \rightarrow [Prop\ (Panel\ ())] \rightarrow IO\ (Panel\ ())
\end{aligned}$$

Overerving

wxHaskell refelcts the OO-framework used by wxWidgets.
Inheritance is modelled by so-called phantom types:

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

Inheritance: Hiërarchy

Unfolding the type synonyms shows the type hiërarchy:

$$\begin{aligned} \mathit{Button}\ () &\simeq \mathit{Control}\ (\mathit{CButton}\ ()) \\ &\simeq \mathit{Window}\ (\mathit{CControl}\ (\mathit{CButton}\ ())) \\ &\simeq \mathit{Object}\ (\mathit{CWindow}\ (\mathit{CControl}\ (\mathit{CButton}\ ()))) \end{aligned}$$

Inheritance: Subtyping

$$\textbf{type}\ Frame\ a = Window\ (CFrame\ a)$$

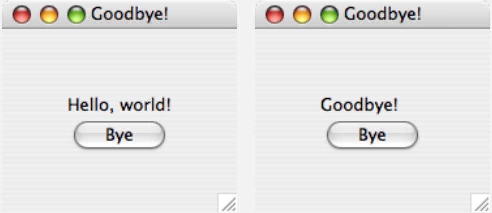
$$\begin{aligned} &frame :: [Prop\ (Frame\ ())] \rightarrow IO\ (Frame\ ()) \\ &button :: Window\ a \rightarrow [Prop\ (Button\ ())] \rightarrow IO\ (Button\ ()) \end{aligned}$$

The function *button* may be called with a value of any subtype of *Window*, so also can be passed a *Frame*:

$$\begin{aligned} gui = \textbf{do}\ & f \leftarrow frame\ [\,] \\ & q \leftarrow button\ f\ [\,] \\ & \ldots \end{aligned}$$

Goodbye!

Our second program:



- If we press the button the welcome message becomes a goodbye message.
- If we press again the window is closed.

Goodbye!: Initialisatone

```haskell
import Graphics.UI.WX
main :: IO ()
main = start goodbye
```

Goodbye!: Interface construction

```
goodbye :: IO ()
goodbye = do f ← frame          [text := "Goodbye!"]
             p ← panel      f   []
             t ← staticText p   [text := "Hello, world!"]
             q ← button     p   [text := "Bye"]
             set q [on command := bye f t q]
             set f [layout     := container p
                                  $ margin 50
                                  $ column 5
                                  $ [centre (widget t)
                                    ,centre (widget q)
                                    ]]
```

Goodbye!: Event handler

$$
\begin{aligned}
&\mathit{bye} && :: \mathit{Frame}\ () \rightarrow \mathit{StaticText}\ () \rightarrow \mathit{Button}\ () \rightarrow \mathit{IO}\ () \\
&\mathit{bye}\ f\ t\ q && = \textbf{do}\ \mathit{txt} \leftarrow \mathit{get}\ t\ \mathit{text} \\
& && \qquad\quad \textbf{if}\ \mathit{txt} \equiv \texttt{"Hello, world!"} \\
& && \qquad\quad\quad \textbf{then}\ \mathit{set}\ t\ [\mathit{text} := \texttt{"Goodbye!"}] \\
& && \qquad\quad\quad \textbf{else}\ \ \mathit{close}\ f
\end{aligned}
$$

Goodbye!: Demo

Attributen

With every widget type we associate a couple of attributes:

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

Properties

A property is a combination of an attribute and a concrete value:

$$\begin{aligned}&\textbf{data}\ \mathit{Prop}\ w = \ldots\\ &(:=) \quad :: \mathit{Attr}\ w\ a \to a \to \mathit{Prop}\ w\\ &\mathit{set} \quad :: w \to [\mathit{Prop}\ w] \to \mathit{IO}\ ()\\ &\mathit{get} \quad :: w \to \mathit{Attr}\ w\ a \to \mathit{IO}\ a\end{aligned}$$

For example:

$$\begin{aligned}\mathit{gui} = \textbf{do}\ &f \leftarrow \mathit{frame}\ [\mathit{text} := \texttt{"Hello, world"}]\\ &\mathit{txt} \leftarrow \mathit{get}\ f\ \mathit{text}\\ &\mathit{set}\ f\ [\mathit{text} := \mathit{txt} +\!\!+ \texttt{"!"}]\\ &\ldots\end{aligned}$$

Events

Events are objects to which we can connect actions ($IO\ ()$ values):

$$\textbf{data}\ Event\ w\ a = \ \ldots$$

$$\textbf{class}\ Commanding\ w\ \textbf{where}$$
$$\quad command :: Event\ w\ (IO\ ())$$

$$\textbf{instance}\ Commanding\ (Button\ a)\ \textbf{where} \ldots$$

Using *on* an event is promoted to an event-handling attribute:

$$on :: Event\ w\ a \rightarrow Attr\ w\ a$$

For example::

$$\begin{aligned} gui = \textbf{do}\ \ & f \leftarrow frame\ [\,] \\ & q \leftarrow button\ f\ [on\ command := close\ f] \end{aligned}$$

Dynamic event handlers

Event handlers can, just like most of the other attributes be replaced dynamically:.

For example::

$$
\begin{aligned}
&bye :: Frame\ () \rightarrow StaticText\ () \rightarrow Button\ () \rightarrow IO\ () \\
&bye\ f\ t\ q = \textbf{do}\ txt \leftarrow get\ t\ text \\
&\qquad \textbf{if}\ txt \equiv \texttt{"Hello, world!"} \\
&\qquad\quad \textbf{then}\ set\ t\ [text := \texttt{"Goodbye!"}] \\
&\qquad\quad \textbf{else}\ \ close\ f
\end{aligned}
$$

Nicer and more robust:

$$
\begin{aligned}
&bye :: Frame\ () \rightarrow StaticText\ () \rightarrow Button\ () \rightarrow IO\ () \\
&bye\ f\ t\ q = \textbf{do}\ set\ t\ [text \qquad\quad := \texttt{"Goodbye!"}] \\
&\qquad\qquad\quad set\ q\ [on\ command := close\ f]
\end{aligned}
$$

Bouncing balls:Demo

Bouncing Balls: Initialisation

```
import Graphics.UI.WX
main :: IO ()
main = start balls
```

Bouncing Balls: Constants

Width and heigth of the screen; radius of a ball:

$$
\begin{aligned}
&\mathit{width}, \mathit{height}, \mathit{radius} :: \mathit{Int} \\
&\mathit{width} = 300 \\
&\mathit{height} = 300 \\
&\mathit{radius} = 10
\end{aligned}
$$

Maximal y-coördinate of a bal:

$$
\begin{aligned}
&\mathit{maxH} :: \mathit{Int} \\
&\mathit{maxH} = \mathit{height} - \mathit{radius}
\end{aligned}
$$

Stuiterballen

A point consists of an x- and a y-coördinate:

$$\textbf{data}\ Point = Point\ Int\ Int$$

We represent a bal by a list of future positions:

$$
\begin{aligned}
&\textbf{type}\ Ball && = [Point] \\
&bouncing && :: Point \rightarrow Ball \\
&bouncing\ (Point\ x\ y) && = \textbf{let}\ hs = bounce\ (maxH - y)\ 0 \\
& && \quad\ \textbf{in}\ \ [Point\ x\ (maxH - h) \mid h \leftarrow hs] \\
&bounce && :: Int \rightarrow Int \rightarrow [Int] \\
&bounce\ h\ v && \\
&\quad \mid h \leqslant 0 \wedge v \equiv 0 && = replicate\ 20\ 0 \\
&\quad \mid h \leqslant 0 \wedge v < 0 && = bounce\ 0\ ((-v) - 2) \\
&\quad \mid otherwise && = h : bounce\ (h + v)\ (v - 1)
\end{aligned}
$$

Bouncing balls: Drawing the scene

We can draw on a device context:

$$\textbf{type}\ DC\ a = \ldots$$
$$circle :: DC\ a \rightarrow Point \rightarrow Int \rightarrow [Prop\ (DC\ a)] \rightarrow IO\ ()$$

$$drawBall :: DC\ a \rightarrow Point \rightarrow IO\ ()$$
$$drawBall\ dc\ pt = circle\ dc\ pt\ radius\ [\,]$$

Bouncing Balls: Interface Construction

```
balls :: IO ()
balls = do
  vballs ← variable    [value      := []]
  f      ← frameFixed  [text       := "Bouncing balls"]
  p      ← panel f     [on paint   := paintBalls vballs
                       ,bgcolor    := white]
  t      ← timer f     [interval   := 20
                       ,on command := next vballs p]
  set p [on click := dropBall vballs]
  set f [layout   := minsize (sz width height)
                     $ widget p]
```

Bouncing Balls: Event handler for (Re)Drawing

Ieder *Window* heeft een *paint*-*Event*:

```
class Paint w where ...
instance Paint (Window a) where ...
paint :: (Paint w) ⇒ Event w (DC ()→Rect→IO ())
```

Teken elke bal op zijn eerstvolgende positie:

```
paintBalls                 :: Var [Ball]→DC a→Rect→IO ()
paintBalls vballs dc vw = do
  bs ← get vballs value
  set dc [brushColor := red, brushKind := BrushSolid]
  sequence_ [drawBall dc pt | (pt : _) ← bs]
```

Bouncing Balls: Event handler for timer

In setting properties we can access the current value of the properties:

$(:\sim) :: Attr\ w\ a \to (a \to a) \to Prop\ w$

For each ball we get the next position:

$$
\begin{aligned}
&next :: Var\ [Ball] \to Panel\ () \to IO\ () \\
&next\ vballs\ p = \mathbf{do} \\
&\quad set\ vballs\ [value :\sim filter\ (\neg \circ null) \circ map\ tail] \\
&\quad repaint\ p
\end{aligned}
$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Bouncing Balls: Event handler for mouse clicks

Elk *Window* heeft een *click-Event*:

$$\textbf{class}\ \mathit{Reactive}\ w\ \textbf{where}\ldots$$
$$\textbf{instance}\ \mathit{Reactive}\ (\mathit{Window}\ a)\ \textbf{where}\ldots$$
$$\mathit{click} :: (\mathit{Reactive}\ w) \Rightarrow \mathit{Event}\ w\ (\mathit{Point} \rightarrow \mathit{IO}\ ())$$

Laat een bal los op de aangegeven positie:

$$\begin{aligned} &\mathit{dropBall} && :: \mathit{Var}\ [\mathit{Ball}] \rightarrow \mathit{Point} \rightarrow \mathit{IO}\ () \\ &\mathit{dropBall}\ \mathit{vballs}\ \mathit{pt} && = \mathit{set}\ \mathit{vballs}\ [\mathit{value} :\sim (\mathit{bouncing}\ \mathit{pt}:)] \end{aligned}$$

properties: use

Properties are used when constructing new widgets:

$$button :: Window\ a \rightarrow [Prop\ (Button\ ())] \rightarrow IO\ (Button\ ())$$

$$q \leftarrow button\ f\ [text := \texttt{"Quit"}]$$

using *get* en *set*:

$$get :: w \rightarrow Attr\ w\ a \rightarrow IO\ a$$
$$set :: w \rightarrow [Prop\ w] \rightarrow IO\ ()$$

$$x \leftarrow get\ t\ interval$$
$$set\ t\ [interval := x]$$

Property Constructors: Assigment

$(:=)$ is an infix-constructor function:

$(:=) :: Attr\ w\ a \rightarrow a \rightarrow Prop\ w$

Had we assigned the name *Assign* to this constructor we would have written:

$[Assign\ interval\ 20]$

of

$[interval\ `Assign`\ 20]$

instead of the more usual notation:

$[interval := 20]$

Property Constructors: Updates

Yet another infix-constructor:

$$(:\sim) :: Attr\ w\ a \rightarrow (a \rightarrow a) \rightarrow Prop\ w$$

Compare:

$$x \leftarrow get\ t\ interval$$
$$set\ t\ [interval := x + 1]$$

with

$$set\ t\ [interval :\sim succ]$$

Mutable variables

When implementing the bouncing balls examples we used mutable variables:

$$
\begin{aligned}
&balls = \textbf{do} \\
&\quad vballs \leftarrow variable\ [value := [\,]] \\
&\quad \ldots \\
&\quad p \qquad \leftarrow panel\ f\ [on\ paint \qquad := paintBalls\ vballs] \\
&\quad t \qquad \leftarrow timer\ f\ [on\ command := next\ vballs] \\
&\quad \ldots \\
&drawBalls\ vballs = \textbf{do} \\
&\quad bs \leftarrow get\ vballs\ value \\
&\quad \ldots \\
&next\ vballs = set\ vballs\ [value :\sim filter\ (\neg \circ null) \circ map\ tail]
\end{aligned}
$$

Actions on Variables

To preserve referential transparency,operation on mutable variables are of type $IO \ldots$:

$$variabele :: [Prop\ (Var\ a)] \rightarrow IO\ (Var\ a)$$

A variabele holding a value of type a has type $Var\ a$:

$$\textbf{class}\ Valued\ w\ \textbf{where}$$
$$\quad value :: Attr\ (w\ a)\ a$$
$$\textbf{instance}\ Valued\ Var\ \textbf{where} \ldots$$

Imperative Programming: Fibonacci-function

An elegant, but inefficiënt Fibonacci-function:

$$
\begin{aligned}
&\mathit{fib} && :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ n \mid n < 2 && = n \\
&\qquad\ \ \mid \mathit{otherwise} && = \mathit{fib}\ (n-2) + \mathit{fib}\ (n-1)
\end{aligned}
$$

An efficiënt alternative:

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ n = \mathit{fibs}\ !!\ n \\
&\quad \mathbf{where} \\
&\qquad \mathit{fibs} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})
\end{aligned}
$$

Imperative programming: Imperatieve Fibonacci-function

An imperatieve variant (but still proper Haskell):

```
fib       :: Int → IO Int
fib n     = do x ← variable [value := 0]
               y ← variable [value := 1]
               for [1 .. n] $ \_ → do
                 u ← get x value
                 v ← get y value
                 set x [value := v]
                 set y [value := u + v]
               get x value
               return x

for       :: [a] → (a → IO b) → IO [b]
for xs f  = sequence (map f xs)
```

And now Efficient Haskell

$$
\begin{aligned}
&\mathit{fib}\ n = \mathit{fib}'\ n\ 1\ 0 \\
&\quad \textbf{where}\ \mathit{fib}'\ 0\ x\ y = y \\
&\qquad\quad\ \ \mathit{fib}'\ 1\ x\ y = x \\
&\qquad\quad\ \ \mathit{fib}'\ n\ x\ y = \mathit{fib}'\ (n-1)\ (x+y)\ x
\end{aligned}
$$

Pay attention!

Why can't we just write:

- $set\ x\ [value := get\ y\ value]$

Pay attention!

Why can't we just write:

$$set\ x\ [value := get\ y\ value]$$

What is the type of *get y value*?

Pay attention!

Why can't we just write:

$$set\ x\ [value := get\ y\ value]$$

What is the type of *get y value*?

$$get\ y\ value :: IO\ Int$$

and thus not just *Int*, as you might have expected. Reading a variable really is an effect, hence *IO*.

Pay attention!

Why can't we just write:

> $set\ x\ [value := get\ y\ value]$

What is the type of $get\ y\ value$?

> $get\ y\ value :: IO\ Int$

and thus not just Int, as you might have expected. Reading a variable really is an effect, hence IO.
Who understands what `f (x++) + g (x--)` means in C or C++?

Layout-combinators

Widgets like *Frame* () and *Panel* () have an attribuut *layout*:

$$layout :: (Form\ w) \Rightarrow Attr\ w\ Layout$$

For example:

$$\begin{aligned}
&main = start\ gui \\
&gui \quad = \mathbf{do}\ f \leftarrow frame\ [\,] \\
&\qquad\qquad\quad q \leftarrow button\ f\ [on\ command := close\ f\,] \\
&\qquad\qquad\quad set\ f\ [layout := widget\ q\,]
\end{aligned}$$

Combinatoren

Layoutscan be specified using a special set of combinators:

- Embedded: just Haskell-functins
- Defined in *Graphics.UI.WXCore.Layout* and *Graphics.UI.WX.Layout* (zie documentatie)
- Re-exported by *Graphics.UI.WX*

Building blocks

Primitive layouts:

$$
\begin{aligned}
&\mathit{label} &&:: \mathit{String} && \to \mathit{Layout}\\
&\mathit{space} &&:: \mathit{Int} \to \mathit{Int} && \to \mathit{Layout}\\
&\mathit{rule} &&:: \mathit{Int} \to \mathit{Int} && \to \mathit{Layout}\\
&\mathit{widget} &&:: (\mathit{Widget}\ w) \Rightarrow w && \to \mathit{Layout}
\end{aligned}
$$

Composing layouts:

$$
\begin{aligned}
&\mathit{grid} &&:: \mathit{Int} \to \mathit{Int} \to [[\mathit{Layout}]] \to \mathit{Layout}\\
&\mathit{container} &&:: \mathit{Window}\ a \to \mathit{Layout} \to \mathit{Layout}\\
&\mathit{margin} &&:: \mathit{Int} \to \mathit{Layout} \to \mathit{Layout}
\end{aligned}
$$

Abstractions

With a few primitives and combinators we can already define abstractions:

$$
\begin{aligned}
&\mathit{empty} &&:: \mathit{Layout}\\
&\mathit{empty} &&= \mathit{space}\ 0\ 0\\
&\mathit{hrule}, \mathit{vrule} &&:: \mathit{Int} \to \mathit{Layout}\\
&\mathit{hrule}\ n &&= \mathit{rule}\ n\ 1\\
&\mathit{vrule}\ n &&= \mathit{rule}\ 1\ n\\
&\mathit{row}, \mathit{column} &&:: \mathit{Int} \to [\mathit{Layout}] \to \mathit{Layout}\\
&\mathit{row}\ n\ \mathit{ls} &&= \mathit{grid}\ n\ 0\ [\mathit{ls}]\\
&\mathit{column}\ n\ \mathit{ls} &&= \mathit{grid}\ 0\ n\ [[l] \mid l \leftarrow \mathit{ls}]
\end{aligned}
$$

Filling empty space

What to do if a layout does not consume all available space?

- Alignment: wher does a layout show up?
- Expansion: how large will the layout be?
- Stretch: in which direction will the layout strech?

Alingning

Position in the xtra space:

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

valignTop    :: Layout → Layout
valignBottom :: Layout → Layout
valignCenter :: Layout → Layout
```

Extend

Filling the extra space:

$$
\begin{aligned}
&rigid &&:: Layout \rightarrow Layout && \text{-- default}\\
&shaped &&:: Layout \rightarrow Layout && \text{-- follow parent}\\
&expand &&:: Layout \rightarrow Layout && \text{-- fill and extend}
\end{aligned}
$$

Strech

Probably reserving extra space:

$$\begin{aligned} &static &&:: Layout \rightarrow Layout \quad \text{-- default} \\ &hstretch &&:: Layout \rightarrow Layout \\ &vstretch &&:: Layout \rightarrow Layout \end{aligned}$$

Only interesting for grids.

Abstraction:

$$\begin{aligned} &stretch &&:: Layout \rightarrow Layout \\ &stretch &&= hstretch \circ vstretch \end{aligned}$$

Standard layouts: Extend and Float

Using these combinators we can program many layout policies:

$$
\begin{aligned}
&\mathit{alignCenter}, \mathit{alignBottomRight} &&:: \mathit{Layout} \to \mathit{Layout} \\
&\mathit{alignCenter} &&= \mathit{halignCenter} \circ \mathit{valignCenter} \\
&\mathit{alignBottomRight} &&= \mathit{halignRight} \circ \mathit{valignBottom} \\
&\mathit{floatCenter}, \mathit{floatBottomRight} &&:: \mathit{Layout} \to \mathit{Layout} \\
&\mathit{floatCenter} &&= \mathit{stretch} \circ \mathit{alignCenter} \\
&\mathit{floatBottomRight} &&= \mathit{stretch} \circ \mathit{alignBottomRight}
\end{aligned}
$$

Standaard layouts: Filling and Glueing en lijmen

More layout-patterns:

$$
\begin{aligned}
&hfill, vfill, fill &&:: Layout \to Layout \\
&hfill &&= hstretch \circ expand \\
&vfill &&= vstretch \circ expand \\
&fill &&= hfill \circ vfill \\
&hglue, vglue, glue &&:: Layout \to Layout \\
&hglue &&= hstretch\ empty \\
&vglue &&= vstretch\ empty \\
&glue &&= stretch\ empty
\end{aligned}
$$

Layout Demo

Layout Demo

$$main :: IO\ ()$$
$$main = start\ layoutDemo$$

Layout Demo: Widgets

$$
\begin{array}{l}
\mathit{layoutDemo} :: \mathit{IO}\ () \\
\mathit{layoutDemo} = \textbf{do} \\
\quad f^{\wedge\wedge\wedge} \leftarrow \mathit{frame} \quad [\mathit{text} := \texttt{"Layout Demo"}] \\
\quad p \leftarrow \mathit{panel}\ f\ [\,] \\
\quad x \leftarrow \mathit{entry}\ p\ [\mathit{text} := \texttt{"100"}] \\
\quad y \leftarrow \mathit{entry}\ p\ [\mathit{text} := \texttt{"100"}] \\
\quad \mathit{ok} \leftarrow \mathit{button}\ p\ [\mathit{text} := \texttt{"Ok"}] \\
\quad \mathit{can} \leftarrow \mathit{button}\ p\ [\mathit{text} := \texttt{"Cancel"}] \\
\quad \ldots
\end{array}
$$

Layout Demo: Layout

```
layoutDemo :: IO ()
layoutDemo = do
    ...
    set f [layout := container p $ margin 5 $
              column 10 [hfill $ space 0 20,
                         hfill $ hrule 0,
                         margin 10 $ grid 5 5
                           [[label "x", hfill (widget x)],
                            [label "y", hfill (widget y)]],
                         hfill $ hrule 0,
                         hfill $ space 0 20,
                         floatBottomRight $ row 5
                           [widget ok, widget can]]]
```