COMP4075/G54RFP: Lecture 11 & 12

The Threepenny GUI Toolkit

Henrik Nilsson

University of Nottingham, UK

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML ***Document Object Model*** (DOM) to be manipulated

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML ***Document Object Model*** (DOM) to be manipulated
  - handles JavaScript events in Haskell

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML ***Document Object Model*** (DOM) to be manipulated
  - handles JavaScript events in Haskell
- Works by sending JavaScript code to the client.

What is Threepenny (2)

- Frequent communication between browser and server: Threepenny is best used running on localhost or over the local network.

What is Threepenny (2)

- Frequent communication between browser and server: Threepenny is best used running on localhost or over the local network.
- Written by Heinrich Apfelmus.

Rich API

- Full set of widgets (buttons, menus, etc.)
- Drag and Drop
- HTML elements
- Support for CSS
- Canvas for general drawing
- Functional Reactive Programming (FRP)

Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.

Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.
- Set up event handlers to act on events from the elements.

Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.
- Set up event handlers to act on events from the elements.
- Knowing a bit of HTML helps.

The *UI* Monad

Most work take place in the the ***User Interface*** monad UI:

The UI Monad

Most work take place in the the ***User Interface*** monad UI:

- Wrapper around IO; keeps track of e.g. window context.

The UI Monad

Most work take place in the the ***User Interface*** monad UI:

- Wrapper around IO; keeps track of e.g. window context.
- Instance of MonadIO, meaning that any IO operation can be lifted into UI:

$$liftIO :: IO \ \ a \to UI \ \ a$$

The Browser $Window$

- Type $Window$ represents a browser window.

The Browser $Window$

- Type $Window$ represents a browser window.
- It has an attribute $title$ that may be written:

$$title :: WriteAttr\ Window\ String$$

The Browser Window

- Type Window represents a browser window.
- It has an attribute title that may be written:

$$\mathit{title} :: \mathit{WriteAttr}\ \mathit{Window}\ \mathit{String}$$

- Retrieving the current window context:

$$\mathit{askWindow} :: \mathit{UI}\ \mathit{Window}$$

The Browser *Window*

- Type *Window* represents a browser window.
- It has an attribute *title* that may be written:

$$title :: WriteAttr\ Window\ String$$

- Retrieving the current window context:

$$askWindow :: UI\ Window$$

- Window passed to GUI code when server started:

$$startGUI :: Config \to (Window \to UI\ ()) \to IO\ ()$$

Elements

DOM made up of elements:

$$mkElement :: String \to UI\ Element$$

An element ***created*** when action run. Argument is an HTML elemen name: `"div"`, `"h1"`, `"p"`, etc.

Elements

DOM made up of elements:

$$mkElement :: String \rightarrow UI\ Element$$

An element ***created*** when action run.
Argument is an HTML elemen name:
`"div"`, `"h1"`, `"p"`, etc.

Standard elements predefined:

$$\begin{aligned} div &:: UI\ Element \\ h1 &:: UI\ Element \\ br &:: UI\ Element \\ button &:: UI\ Element \end{aligned}$$

Attributes (1)

Elements and other entities like windows have attributes that can be read and written:

$$\textbf{type}\ \mathit{Attr}\ x\ a = \mathit{ReadWriteAttr}\ x\ a\ a$$
$$\textbf{type}\ \mathit{WriteAttr}\ x\ i = \mathit{ReadWriteAttr}\ x\ i\ ()$$
$$\textbf{type}\ \mathit{ReadAttr}\ x\ o = \mathit{ReadWriteAttr}\ x\ ()\ o$$

$$\mathit{set} :: \mathit{ReadWriteAttr}\ x\ i\ o \to i \to \mathit{UI}\ x \to \mathit{UI}\ x$$

$$\mathit{get} :: \mathit{ReadWriteAttr}\ x\ i\ o \to x \to \mathit{UI}\ o$$

$\mathit{ReadWriteAttr}$, $\mathit{WriteAttr}$ etc. are records of functions for attribute reading and/or writing.

set and get work for any type of entity.

Attributes (2)

Sample attributes:

$$
\begin{aligned}
&\mathit{title} :: \mathit{WriteAttr}\ \mathit{Window}\ \mathit{String}\\
&\mathit{color} :: \mathit{WriteAttr}\ \mathit{Element}\ \mathit{String}\\
&\mathit{children} :: \mathit{WriteAttr}\ \mathit{Element}\ [\mathit{Element}]\\
&\mathit{value} :: \mathit{Attr}\ \mathit{Element}\ \mathit{String}\\
&(\#+) :: \mathit{UI}\ \mathit{Element} \to [\mathit{UI}\ \mathit{Element}] \to \mathit{UI}\ \mathit{Element}\\
&(\#.) :: \mathit{UI}\ \mathit{Element} \to \mathit{String} \to \mathit{UI}\ \mathit{Element}
\end{aligned}
$$

$(\#+)$ appends children to a DOM element.
$(\#.)$ sets the CSS class.

Attributes (3)

Example usage ($(\#)$ is reverse function application):

$$
\begin{array}{l}
\mathit{mkElement}\ \texttt{"div"} \\
\quad \#\ \mathit{set}\ \mathit{style} \quad [(\texttt{"color"}, \texttt{"\#CCAABB"})] \\
\quad \#\ \mathit{set}\ \mathit{draggable}\ \mathit{True} \\
\quad \#\ \mathit{set}\ \mathit{children}\ \mathit{otherElements}
\end{array}
$$

Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.

Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.
- Semantically: $Event\ a \approx [(Time, a)]$

Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.
- Semantically: $Event\ a \approx [(Time, a)]$
- $Event$ is an instance of $Functor$.

Events (1)

- The type $\mathit{Event}\ a$ represents streams of time-stamped events carrying values of type a.
- Semantically: $\mathit{Event}\ a \approx [(\mathit{Time}, a)]$
- Event is an instance of $\mathit{Functor}$.
- Event is ***not*** an instance of $\mathit{Applicative}$. The type for $\circledast$ would be

$$\mathit{Event}\ (a \to b) \to \mathit{Event}\ a \to \mathit{Event}\ b$$

However, this makes no sense as event streams in general are not synchronised.

Events (2)

Most events originate from UI elements; e.g.:

- $valueChange :: Element \rightarrow Event\ String$
- $click :: Element \rightarrow Event\ ()$
- $mousemove :: Element \rightarrow Event\ (Int, Int)$
  (coordinates relative to the element)
- $hover :: Element \rightarrow Event\ ()$
- $focus :: Element \rightarrow Event\ ()$
- $keypress :: Element \rightarrow Event\ Char$

Events (3)

One or more handlers can be registered for events:

$$register :: Event\ a \rightarrow Handler\ a \rightarrow IO\ (IO\ ())$$

The resulting action is intended for deregistering a handler; future functionality.

Events (4)

Usually, registration is done using convenience functions designed for use directly with elements and in the UI monad:

$$on :: (element \to Event\ a)$$
$$\to element \to (a \to UI\ void) \to UI\ ()$$

For example:

$$\mathbf{do}$$
$$\ldots$$
$$on\ click\ element\ \$\ \lambda\_ \to \ldots$$
$$\ldots$$

Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.

Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.
- Semantically: $Behavior\ a \approx Time \rightarrow a$

Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.
- Semantically: $Behavior\ a \approx Time \to a$
- $Behavior$ is an instance of $Functor$ ***and*** $Applicative$.

Behaviors (1)

- The type $\mathit{Behavior}\ a$ represents continuously time-varying values of type a.
- Semantically: $\mathit{Behavior}\ a \approx \mathit{Time} \to a$
- $\mathit{Behavior}$ is an instance of $\mathit{Functor}$ **and** $\mathit{Applicative}$.
- Recall that events are not an applicative. However, the following provides similar functionality:

$$
\begin{aligned}
(\text{<@>}) :: \mathit{Behavior}\ (a \to b) \\
\to \mathit{Event}\ a \to \mathit{Event}\ b
\end{aligned}
$$

Behaviors (2)

- Attributes can be set to time-varying values:

$$\begin{aligned} &sink :: ReadWriteAttr\ x\ i\ o \\ &\qquad \to Behavior\ i \to UI\ x \to UI\ x \end{aligned}$$

Behaviors (2)

- Attributes can be set to time-varying values:

$$\begin{aligned}&\mathit{sink} :: \mathit{ReadWriteAttr}\ x\ i\ o\\&\qquad\qquad \to \mathit{Behavior}\ i \to \mathit{UI}\ x \to \mathit{UI}\ x\end{aligned}$$

- There is also:

$$\begin{aligned}&\mathit{onChanges} :: \mathit{Behavior}\ a\\&\qquad\qquad\qquad \to (a \to \mathit{UI}\ \mathit{void}) \to \mathit{UI}\ ()\end{aligned}$$

But conceptually questionable as a behavior in general is ***always*** changing.

FRP (1)

Threepenny offers support for Functional Reactive Programming (FRP): transforming and composing behaviours and events as “whole values”.

FRP (1)

Threepenny offers support for Functional Reactive Programming (FRP): transforming and composing behaviours and events as “whole values”.

For example:

- $\mathit{filterJust} :: \mathit{Event}\ (\mathit{Maybe}\ a) \rightarrow \mathit{Event}\ a$
- $\mathit{unionWith} :: (a \rightarrow a \rightarrow a)$
  $\rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a$
- $\mathit{unions} :: [\mathit{Event}\ a] \rightarrow \mathit{Event}\ [a]$
- $\mathit{split} :: \mathit{Event}\ (\mathit{Either}\ a\ b) \rightarrow (\mathit{Event}\ a, \mathit{Event}\ b)$

FRP (2)

- $accumE :: MonadIO\ m$
  $\quad \Rightarrow a \to Event\ (a \to a) \to m\ (Event\ a)$
- $accumB :: MonadIO\ m$
  $\quad \Rightarrow a \to Event\ (a \to a) \to m\ (Behavior\ a)$
- $stepper :: MonadIO\ m$
  $\quad \Rightarrow a \to Event\ a \to m\ (Behavior\ a)$
- $(\lessdot\!@\!\gtrdot) :: Behavior\ (a \to b)$
  $\quad \to Event\ a \to Event\ b$

FRP (2)

- $accumE :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ (a \rightarrow a) \rightarrow m\ (Event\ a)$
- $accumB :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ (a \rightarrow a) \rightarrow m\ (Behavior\ a)$
- $stepper :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ a \rightarrow m\ (Behavior\ a)$
- $(\lessdot @ \gtrdot) :: Behavior\ (a \rightarrow b)$
  $\rightarrow Event\ a \rightarrow Event\ b$

Note: Stateful events and behaviors are returned as monadic computations.

Hello World (1)

A simple “Hello World” example:

Hello World (1)

A simple “Hello World” example:

- Display a button

Hello World (1)

A simple “Hello World” example:

- Display a button
- Change its text when clicked

Hello World (1)

A simple "Hello World" example:

- Display a button
- Change its text when clicked

First import the module. Large API, so partly qualified import recommended:

$$\textbf{module}\ \mathit{Main}\ \textbf{where}$$

$$\begin{array}{lll}\textbf{import} & \mathit{qualified} & \mathit{Graphics.UI.Threepenny\ as\ UI} \\ \textbf{import} & & \mathit{Graphics.UI.Threepenny.Core}\end{array}$$

Hello World (2)

The $startGUI$ function starts a server:

$$startGUI :: Config \to (Window \to UI\ ()) \to IO\ ()$$

- $Config$-records carry configuartion parameters.
- $Window$ represents a browser window.
- The function $Window \to UI\ ()$ is called whenever a browser connects to the server and builds the initial HTML page.

Hello World (3)

Start a server listening on port 8023;
static content served from `../wwwroot`:

$$
\begin{array}{l}
\mathit{main} :: \mathit{IO}\ ()\\
\mathit{main} = \mathbf{do}\\
\quad \mathit{startGUI}\\
\qquad \mathit{defaultConfig}\\
\qquad\quad \{\mathit{jsPort} \quad = \mathit{Just}\ 8023,\\
\qquad\qquad \mathit{jsStatic} = \mathit{Just}\ \texttt{"../wwwroot"}\}\\
\qquad \mathit{setup}
\end{array}
$$

Hello World (4)

Start by setting the window title:

$$setup :: Window \rightarrow UI\ ()$$
$$setup\ window = \mathbf{do}$$
$$\quad return\ window\ \#\ set\ UI.title\ \texttt{"Hello World!"}$$

Reversed function application: $(\#) :: a \rightarrow (a \rightarrow b) \rightarrow b$

set has type:

$$set :: ReadWriteAttr\ x\ i\ o \rightarrow i \rightarrow UI\ x \rightarrow UI\ x$$

The window reference is a pure value, passed in, hence the need to lift it into a UI computation using $return$.

Hello World (5)

Then create a button element:

$$button \leftarrow UI.button \;\#\; set\ UI.text\ \texttt{"Click me!"}$$

Hello World (5)

Then create a button element:

$$button \leftarrow UI.button \# set\ UI.text\ \texttt{"Click me!"}$$

Note that $UI.button$ has type:

$$button :: UI\ Element$$

A new button is is ***created*** whenever that action is run.

Hello World (5)

Then create a button element:

$$button \leftarrow UI.button \# set\ UI.text\ \texttt{"Click me!"}$$

Note that $UI.button$ has type:

$$button :: UI\ \ Element$$

A new button is is ***created*** whenever that action is run.

DOM elements can be accessed much like in JavaScript: searched, updated, moved, inspected.

Hello World (6)

To display the button, it must be attached to the DOM:

$$getBody\ window\ \#+\ [\,element\ \ button\,]$$

Hello World (6)

To display the button, it must be attached to the DOM:

$$getBody\ window \mathbin{\#\!\!+} [element\ button]$$

The combinator $(\#+)$ appends DOM elements as children to a given element:

$$\begin{aligned}(\#+) :: UI\ Element \to [UI\ Element]\\ \to UI\ Element\end{aligned}$$

Hello World (6)

To display the button, it must be attached to the DOM:

$$getBody\ window \mathbin{\#+} [element\ button]$$

The combinator $(\#+)$ appends DOM elements as children to a given element:

$$(\#+) :: UI\ Element \to [UI\ Element] \to UI\ Element$$

$getBody$ gets the body DOM element:

$$getBody :: Window \to UI\ Element$$

Here, $element$ is just $return$.

Hello World (7)

Finally, register an event handler for the click event to change the text of the button:

$$
\begin{aligned}
&\mathit{on}\ \mathit{UI}.\mathit{click}\ \mathit{button}\ \$\ \mathit{const}\ \$\ \mathbf{do} \\
&\quad \mathit{element}\ \mathit{button} \\
&\qquad \#\ \mathit{set}\ \mathit{UI}.\mathit{text}\ \texttt{"I have been clicked!"}
\end{aligned}
$$

Types:

$$
\begin{aligned}
&\mathit{on} :: (\mathit{element} \to \mathit{Event}\ a) \to \mathit{element} \\
&\qquad \to (a \to \mathit{UI}\ \mathit{void}) \to \mathit{UI}\ () \\
&\mathit{UI}.\mathit{click} :: \mathit{Element} \to \mathit{Event}\ ()
\end{aligned}
$$

Buttons (1)

$$mkButton :: String \to UI\ (Element, Element)$$
$$mkButton\ title = \mathbf{do}$$
$$\quad button \leftarrow UI.button\ \#.\ \texttt{"button"}\ \#+\ [string\ title]$$
$$\quad view \leftarrow UI.p\ \#+\ [element\ button]$$
$$\quad return\ (button, view)$$
$$mkButtons :: UI\ [Element]$$
$$mkButtons = \mathbf{do}$$
$$\quad list \leftarrow UI.ul\ \#.\ \texttt{"buttons-list"}$$
$$\quad \ldots$$

Buttons (2)

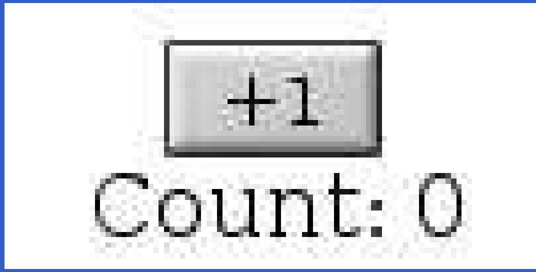
```
(button1, view1) ← mkButton button1Title
on UI.hover button1 $ \_ → do
  element button1 # set text (button1Title ++ " [hover]")
on UI.leave button1 $ \_ → do
  element button1 # set text button1Title
on UI.click button1 $ \_ → do
  element button1 # set text (button1Title ++ " [pressed]"
  liftIO $ threadDelay $ 1000 * 1000 * 1
  element list
    #+ [UI.li # set html "<b>Delayed</b> result!"]
```

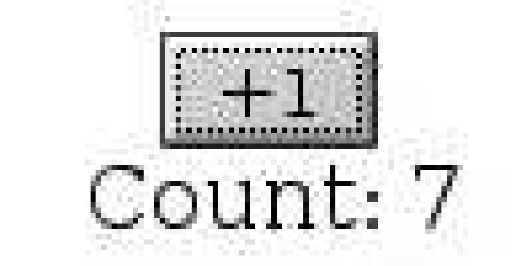
Buttons (3)

```
(button2, view2) ← mkButton button2Title
on UI.hover button2 $ \_ → do
  element button2 # set text (button2Title ++ " [hover]")
on UI.leave button2 $ \_ → do
  element button2 # set text button2Title
on UI.click button2 $ \_ → do
  element button2 # set text (button2Title ++ " [pressed]"
  element list
    #+ [UI.li # set html "Zap! Quick result!"]
return [list, view1, view2]
```

Counter Example 1 (1)

Simple counter, basic imperative style.





Idea:

- Keep the count in an imperative variable
- The click event handler increments the counter and updates the display accordingly.

Counter Example 1 (2)

$$
\begin{array}{l}
\mathit{setup} :: \mathit{Window} \rightarrow \mathit{UI}\ () \\
\mathit{setup}\ \mathit{window} = \mathbf{do} \\
\quad \mathit{return}\ \mathit{window} \\
\qquad \#\ \mathit{set}\ \mathit{UI.title}\ \texttt{"Counter Example 1"} \\
\quad \mathbf{let}\ \mathit{initCount} = 0 \\
\quad \mathit{counter} \leftarrow \mathit{liftIO}\ \$\ \mathit{newIORef}\ \mathit{initCount} \\
\quad \mathit{button}\ \ \leftarrow \mathit{UI.button}\ \#\ \mathit{set}\ \mathit{UI.text}\ \texttt{"+1"} \\
\quad \mathit{label}\ \ \ \ \leftarrow \mathit{UI.label}\ \#\ \mathit{set}\ \mathit{UI.text} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{"Count: "} +\!\!+ \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{show}\ \mathit{initCount})
\end{array}
$$

Counter Example 1 (3)

$$
\begin{array}{l}
\mathit{getBody}\ \mathit{window}\ \#\!+ [\mathit{UI}.\mathit{center} \\
\qquad\qquad\qquad\qquad\qquad\qquad \#\!+ [\mathit{element}\ \mathit{button}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{UI}.\mathit{br}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{element}\ \mathit{label}]] \\
\mathit{on}\ \mathit{UI}.\mathit{click}\ \mathit{button}\ \$\ \mathit{const}\ \$\ \mathbf{do} \\
\quad \mathit{count} \leftarrow \mathit{liftIO}\ \$\ \mathbf{do} \\
\qquad \mathit{modifyIORef}\ \mathit{counter}\ (+1) \\
\qquad \mathit{readIORef}\ \mathit{counter} \\
\quad \mathit{element}\ \mathit{label}\ \#\ \mathit{set}\ \mathit{UI}.\mathit{text}\ (\texttt{"Count: "} +\!\!+ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{show}\ \mathit{count})
\end{array}
$$

Counter Example 2 (1)

Counter with reset, “object-oriented” style.

Idea:

- Make a counter object with encapsulated state and two operations: reset and increment.
- Make a display object with a method for displaying a value.

Counter Example 2 (2)

Make a counter object:

$$
\begin{aligned}
&\mathit{mkCounter} :: \mathit{Int} \to \mathit{UI}\ (\mathit{UI}\ \mathit{Int}, \mathit{UI}\ \mathit{Int})\\
&\mathit{mkCounter}\ \mathit{initCount} = \mathbf{do}\\
&\quad \mathit{counter} \leftarrow \mathit{liftIO}\ \$\ \mathit{newIORef}\ \mathit{initCount}\\
&\quad \mathbf{let}\ \mathit{reset} = \mathit{liftIO}\ \$\ \mathit{writeIORef}\ \mathit{counter}\ \mathit{initCount}\\
&\qquad\qquad\qquad\qquad\qquad \gg \mathit{return}\ \mathit{initCount}\\
&\qquad\quad\ \mathit{incr}\ \ = \mathit{liftIO}\ \$\ \mathit{modifyIORef}\ \mathit{counter}\ (+1)\\
&\qquad\qquad\qquad\qquad\qquad \gg \mathit{readIORef}\ \mathit{counter}\\
&\quad \mathit{return}\ (\mathit{reset}, \mathit{incr})
\end{aligned}
$$

Counter Example 2 (3)

Make a display object:

$$
\begin{array}{l}
mkDisplay :: Int \rightarrow UI\ (Element, Int \rightarrow UI\ ())\\
mkDisplay\ initCount = \mathbf{do}\\
\quad \mathbf{let}\ showCount\ count =\\
\qquad\qquad \texttt{"Count: "} +\!\!+\ show\ count\\
\quad display \leftarrow UI.label\ \#\ set\ UI.text\\
\qquad\qquad\qquad\qquad\qquad (showCount\ initCount)\\
\quad \mathbf{let}\ dispCount\ count =\\
\qquad\qquad ()\ {<}\$ element\ display\\
\qquad\qquad\qquad\qquad \#\ set\ UI.text\ (showCount\ count)\\
\quad return\ (display, dispCount)
\end{array}
$$

Counter Example 2 (4)

$$
\begin{array}{l}
\mathit{setup} :: \mathit{Window} \rightarrow \mathit{UI}\ () \\
\mathit{setup}\ \mathit{window} = \mathbf{do} \\
\quad \mathit{return}\ \mathit{window} \\
\qquad \#\ \mathit{set}\ \mathit{UI}.\mathit{title}\ \texttt{"Counter Example 2"} \\
\quad \mathbf{let}\ \mathit{initCount} = 0 \\
\quad (\mathit{reset}, \mathit{incr}) \leftarrow \mathit{mkCounter}\ \mathit{initCount} \\
\quad (\mathit{display}, \mathit{dispCount}) \leftarrow \mathit{mkDisplay}\ \mathit{initCount} \\
\quad \mathit{buttonRst} \leftarrow \mathit{UI}.\mathit{button}\ \#\ \mathit{set}\ \mathit{UI}.\mathit{text}\ \texttt{"RST"} \\
\quad \mathit{buttonInc} \leftarrow \mathit{UI}.\mathit{button}\ \#\ \mathit{set}\ \mathit{UI}.\mathit{text}\ \texttt{"+1"}
\end{array}
$$

Counter Example 2 (5)

$$
\begin{array}{l}
getBody\ window \\
\quad \#\!\!+ [\, UI.center\ \#\!\!+ [\, element\ buttonRst, \\
\qquad\qquad\qquad\qquad\qquad\quad element\ buttonInc, \\
\qquad\qquad\qquad\qquad\qquad\quad UI.br, \\
\qquad\qquad\qquad\qquad\qquad\quad element\ display]] \\
on\ UI.click\ buttonRst\ \$\ const\ \$\ reset \gg= dispCount \\
on\ UI.click\ buttonInc\ \$\ const\ \$\ incr \gg= dispCount
\end{array}
$$

Counter Example 3 (1)

Counter with reset, FRP style.

Idea:

- Accumulate the button clicks into a ***time-varying*** count; i.e., a $Behavior\ Int$.
- Make the text attribute of the display a time-varying text directly derived from the count; i.e., a $Behavior\ String$.

Counter Example 3 (2)

$$
\begin{array}{l}
setup :: Window \to UI\ () \\
setup\ window = \mathbf{do} \\
\quad return\ window \\
\qquad \#\ set\ UI.title\ \texttt{"Counter Example 3"} \\
\quad \mathbf{let}\ initCount = 0 \\
\quad buttonRst \leftarrow UI.button\ \#\ set\ UI.text\ \texttt{"RST"} \\
\quad buttonInc \leftarrow UI.button\ \#\ set\ UI.text\ \texttt{"+1"} \\
\quad \mathbf{let}\ reset \quad = (const\ 0) <\$ UI.click\ buttonRst \\
\quad \mathbf{let}\ incr \quad\ = (+1) \qquad <\$ UI.click\ buttonInc
\end{array}
$$

Note: *Event* and *Behavior* are instances of *Functor*.

Counter Example 3 (3)

$$
\begin{aligned}
&\mathit{count} \;\; \leftarrow \mathit{accumB}\ 0\ \$\ \mathit{unionWith}\ \mathit{const}\ \mathit{reset}\ \mathit{incr} \\
&\mathit{display} \leftarrow \mathit{UI.label} \\
&\qquad\qquad\qquad \#\ \mathit{sink}\ \mathit{UI.text} \\
&\qquad\qquad\qquad\qquad (\mathit{fmap}\ \mathit{showCount}\ \mathit{count})
\end{aligned}
$$

Type signatures:

$$
\begin{aligned}
&\mathit{accumB} :: \mathit{MonadIO}\ m \Rightarrow \\
&\qquad a \rightarrow \mathit{Event}\ (a \rightarrow a) \rightarrow m\ (\mathit{Behavior}\ a) \\
&\mathit{unionWith} :: (a \rightarrow a \rightarrow a) \\
&\qquad \rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a \\
&\mathit{sink} :: \mathit{ReadWriteAttr}\ x\ i\ o \\
&\qquad \rightarrow \mathit{Behavior}\ i \rightarrow \mathit{UI}\ x \rightarrow \mathit{UI}\ x
\end{aligned}
$$

Counter Example 3 (4)

$$
\begin{array}{l}
\mathit{getBody}\ \mathit{window} \\
\quad \#{+}\ [\mathit{UI}.\mathit{center}\ \#{+}\ [\mathit{element}\ \mathit{buttonRst}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{element}\ \mathit{buttonInc}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{UI}.\mathit{br}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{element}\ \mathit{display}]]
\end{array}
$$

Counter Example 3 (4)

$$
\begin{aligned}
&\mathit{getBody}\ \mathit{window} \\
&\quad \#+ [\mathit{UI}.\mathit{center}\ \#+ [\mathit{element}\ \mathit{buttonRst}, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{element}\ \mathit{buttonInc}, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{UI}.\mathit{br}, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{element}\ \mathit{display}]]
\end{aligned}
$$

- No callbacks.

Counter Example 3 (4)

$$
\begin{aligned}
& getBody\ window \\
& \quad \#+ \left[UI.center \ \#+ \left[\begin{aligned} & element\ buttonRst, \\ & element\ buttonInc, \\ & UI.br, \\ & element\ display \end{aligned} \right] \right]
\end{aligned}
$$

- No callbacks.
- Thus no “callback soup” or “callback hell”!

Counter Example 3 (4)

$$
\begin{aligned}
&getBody\ window \\
&\quad \#{+}\ [\,UI.center\ \#{+}\ [\,element\ buttonRst, \\
&\qquad\qquad\qquad\qquad\qquad\quad element\ buttonInc, \\
&\qquad\qquad\qquad\qquad\qquad\quad UI.br, \\
&\qquad\qquad\qquad\qquad\qquad\quad element\ display\,]\,]
\end{aligned}
$$

- No callbacks.
- Thus no “callback soup” or “callback hell”!
- Fairly declarative description of system: ***Whole-value Programming***.

Counter Example 3 (4)

$$
\begin{array}{l}
getBody\ window \\
\quad \#\!+ [UI.center \#\!+ [element\ buttonRst, \\
\qquad\qquad\qquad\qquad\quad element\ buttonInc, \\
\qquad\qquad\qquad\qquad\quad UI.br, \\
\qquad\qquad\qquad\qquad\quad element\ display]]
\end{array}
$$

- No callbacks.
- Thus no “callback soup” or “callback hell”!
- Fairly declarative description of system: ***Whole-value Programming***.
- This style of programming has had significant impact on programming practice well beyond FP.

Currency Converter (1)

$$return\ window\ \#\ set\ title\ \texttt{"Currency Converter"}$$

$$dollar \leftarrow UI.input$$

$$euro \leftarrow UI.input$$

$$getBody\ window\ \#{+}\ [$$
$$\quad column\ [$$
$$\qquad grid\ [[string\ \texttt{"Dollar:"}, element\ dollar]$$
$$\qquad\qquad ,[string\ \texttt{"Euro:"}, element\ euro]]$$
$$\quad ,string\ \texttt{"Amounts update while typing."}$$
$$\quad]]$$

Currency Converter (2)

$euroIn \leftarrow stepper$ `"0"` $\$ \ UI.valueChange\ euro$

$dollarIn \leftarrow stepper$ `"0"` $\$ \ UI.valueChange\ dollar$

let

$rate = 0.7 :: Double$

$withString\ f =$

$maybe$ `"-"` $(printf$ `"%.2f"`$) \circ fmap\ f \circ readMay$

$dollarOut = withString\ (/rate) < \$ > euroIn$

$euroOut = withString\ (*rate) < \$ > dollarIn$

$element\ euro \# sink\ value\ euroOut$

$element\ dollar \# sink\ value\ dollarOut$

Reading

- Overview, including references to tutorials and examples:

  `http://wiki.haskell.org/Threepenny-gui`

- API reference:

  `http://hackage.haskell.org/package/threepenny-gui`