Functional Reactive Haskellscript

Tim Soethout
Alejandro Serrano

Advanced Functional Programming
Computing Science, Utrecht University

January 27, 2011

Contents

1. Haskell to Javascript
2. Functional Reactive Programming
3. Yampa
4. wxFruit
5. Functional Reactive Haskellscript
6. Conclusion

Haskell to Javascript

- Recently a javascript backend was implemented in the UHC.

[1] As shown by Atze Dijkstra on the CS colloquium of 2010-12-02

Haskell to Javascript

- Recently a javascript backend was implemented in the UHC.
- The basic functionality of Haskell (such as function application) are implented in javascript.[1]

[1] As shown by Atze Dijkstra on the CS colloquium of 2010-12-02

Haskell to Javascript

- Recently a javascript backend was implemented in the UHC.
- The basic functionality of Haskell (such as function application) are implented in javascript.[1]
- The Haskell Foreign Function Interface (FFI) is used to make calls to javascript objects from the Haskell world.

[1] As shown by Atze Dijkstra on the CS colloquium of 2010-12-02

Haskell to Javascript (2)

Demo

Motivation

Our assignment:

- To make a proof of concept using this javascript backend.

Motivation

Our assignment:

- To make a proof of concept using this javascript backend.
- We dove into Functional Reactive Programming and tried to make it work in javascript.

Functional Reactive Programming

- Functional variant of reactive programming.

Functional Reactive Programming

- Functional variant of reactive programming.
- Describes time flow in a declarative/functional programming style.

Functional Reactive Programming

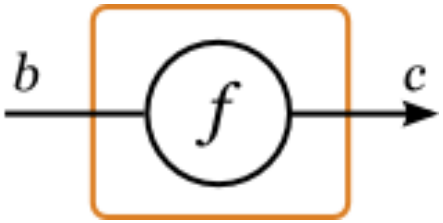
- Functional variant of reactive programming.
- Describes time flow in a declarative/functional programming style.
- We have looked at an implementation: Yampa

Arrows

$$
\begin{aligned}
&\textbf{class}\ \mathit{Arrow}\ a\ \textbf{where}\\
&\quad \mathit{arr} \quad :: (b \to c) \to a\ b\ c\\
&\quad (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d\\
&\quad \mathit{first} \ :: a\ b\ c \to a\ (b, d)\ (c, d)
\end{aligned}
$$

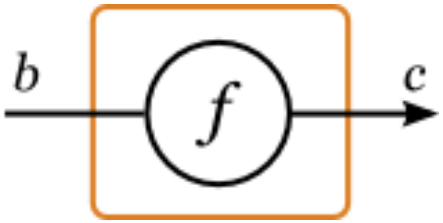
Arrows (2)

- $arr :: (b \to c) \to a\ b\ c$: Each function may be treated as a computation.

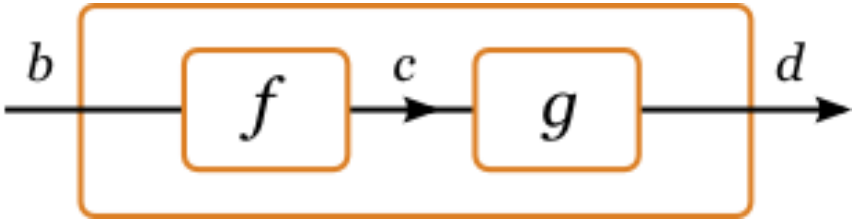


Arrows (2)

- $arr :: (b \to c) \to a\ b\ c$: Each function may be treated as a computation.

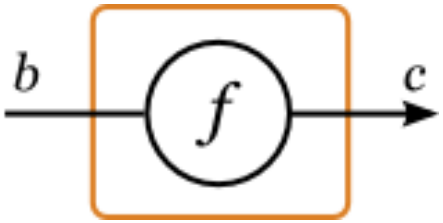


- $(\ggg :: a\ b\ c \to a\ c\ d \to a\ b\ d)$: Computations may be composed, by connecting the output of the first to the input of the second.

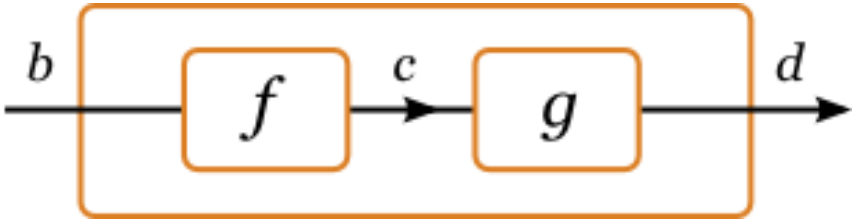


Arrows (2)

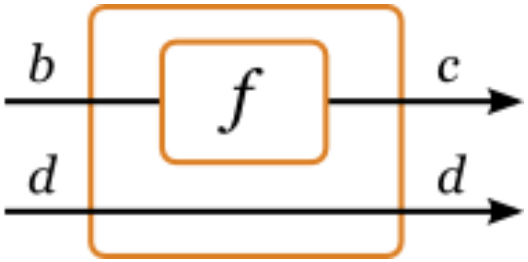
- $arr :: (b \to c) \to a\ b\ c$: Each function may be treated as a computation.



- $(\ggg :: a\ b\ c \to a\ c\ d \to a\ b\ d)$: Computations may be composed, by connecting the output of the first to the input of the second.



- $first :: a\ b\ c \to a\ (b, d)\ (c, d)$: A computation may be applied to part of the input, with the rest copied through to the output.



Arrows (3)

There is a preprocessor that parses arrow syntax into real Haskell, so we can write things like:

$$
\begin{array}{l}
addA :: Arrow\ a \Rightarrow a\ b\ Int \rightarrow a\ b\ Int \rightarrow a\ b\ Int \\
addA\ f\ g = \\
\quad proc\ x \rightarrow \mathbf{do} \\
\qquad y \leftarrow f \prec x \\
\qquad z \leftarrow g \prec x \\
\qquad returnA \prec y + z
\end{array}
$$

Yampa

Signals:

$$Signal\ a = Time \rightarrow a$$

Yampa

Signals:

$$Signal\ a = Time \rightarrow a$$

Signal functions:

$$SF\ a\ b = Signal\ a \rightarrow Signal\ b$$

Yampa

Signals:

$$Signal\ a = Time \rightarrow a$$

Signal functions:

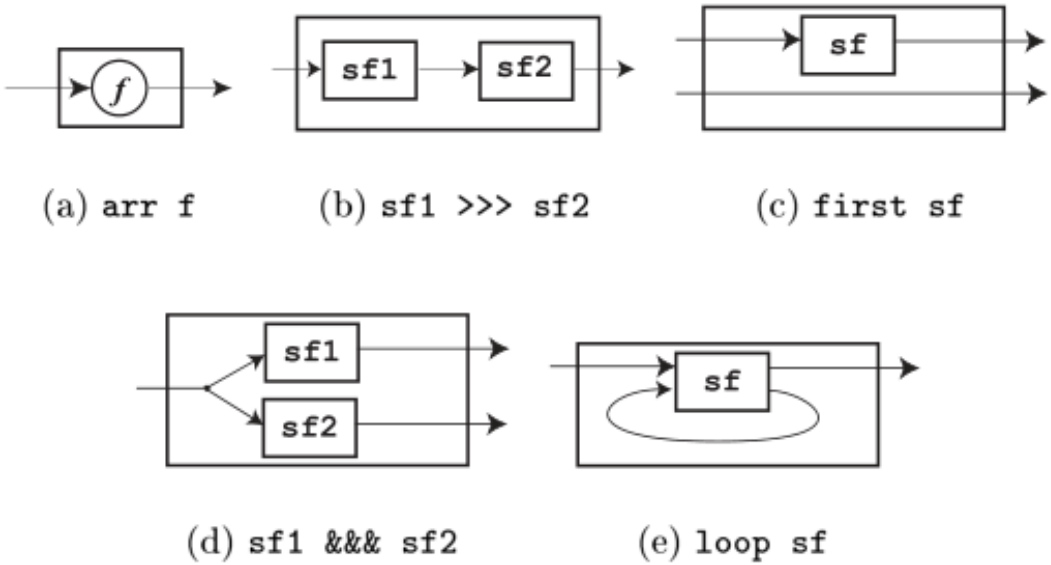
$$SF\ a\ b = Signal\ a \rightarrow Signal\ b$$

Signal functions are an instance of arrow.

Arrow SF

instance $SF\ a\ b$ **where**

$$
\begin{aligned}
&arr && :: (b \to c) \to SF\ b\ c \\
&(\ggg) && :: SF\ b\ c \to SF\ c\ d \to SF\ b\ d \\
&first && :: SF\ b\ c \to SF\ (b, d)\ (c, d)
\end{aligned}
$$



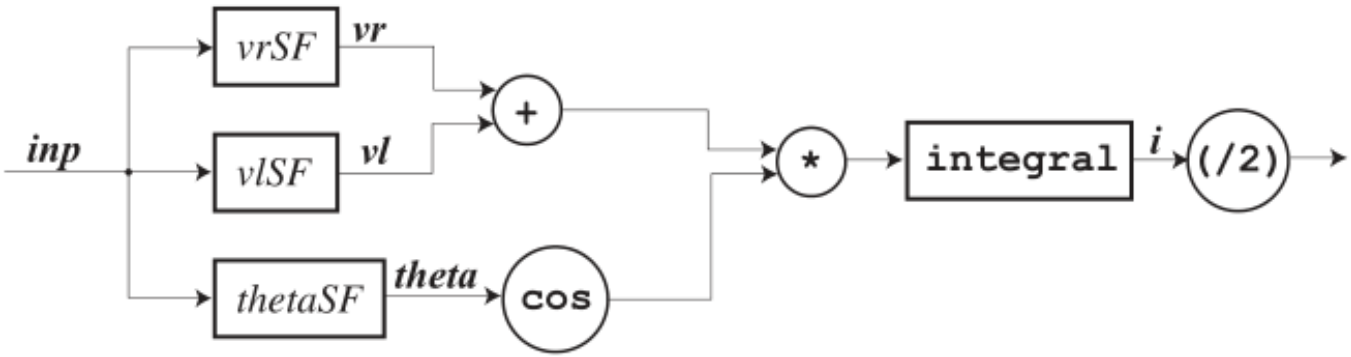
(a) `arr f` (b) `sf1 >>> sf2` (c) `first sf`

(d) `sf1 &&& sf2` (e) `loop sf`

Arrow SF (2)

Example of a Signal Function:

$$
\begin{aligned}
&sf = proc\ inp \rightarrow \mathbf{do} \\
&\quad vr \leftarrow vrSF \prec inp \\
&\quad vl \leftarrow vlSF \prec inp \\
&\quad theta \leftarrow thetaSF \prec inp \\
&\quad i \leftarrow integral \prec (vr + vl) * cos\ theta \\
&\quad returnA \prec (i\ /\ 2)
\end{aligned}
$$



Yampa Space Invaders Example[2]

```
data SimpleGunState = SimpleGunState {
  sgsPos   :: Position2,
  sgsVel   :: Velocity2,
  sgsFired :: Event ()
}
type SimpleGun = SF GameInput SimpleGunState
```

[2] The Yampa Arcade (Haskell'03, August 25, 2003)

Yampa Space Invaders Example (2)

```
simpleGun :: Position2 → SimpleGun
simpleGun (Point2 x0 y0) = proc gi → do
  (Point2 xd _) ← ptrPos ≺ gi
  rec
      -- Controller
    let ad = 10 * (xd − x) − 5 * v
      -- Physics
    v ← integral              ≺ clampAcc v ad
    x ← (x0+)^≪ integral ≺ v
  fire ← leftButtonPress ≺ gi
  returnA ≺ SimpleGunState {
    sgsPos   = (Point2 x y0),
    sgsVel   = (vector2 v 0),
    sgsFired = fire
  }
```

Yampa Space Invaders Example (3)

- In a similar way the missiles and aliens are added.

Yampa Space Invaders Example (3)

- In a similar way the missiles and aliens are added.
- The Yampa function *dpSwitch* is used to switch over all the instances in the world and process them.

wxFruit

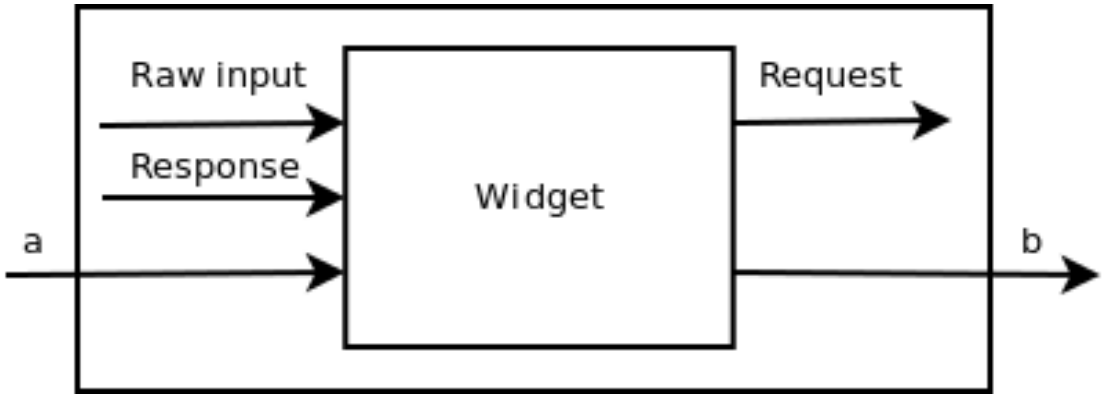
wxFruit was a proof-of-concept for creating GUIs using the *functional reactive* paradigm and *arrow* syntax.

- There were no new releases since February 2009
- But someone upgraded the code to compile in GHC 6.12 and 7.0 in December 2010
- `http://www.haskell.org/haskellwiki/WxFruit`

wxFruit widgets

Each widget is viewed as a Yampa Signal Transformer

$$\begin{aligned}&\textbf{type}\ \mathit{Widget}\ a\ b = \\ &\quad \mathit{SF}\ (\mathit{WXRawInput}, \mathit{Event}\ \mathit{WXWidgetResp}, a) \\ &\qquad (\mathit{Event}\ \mathit{WXWidgetReq}, b)\end{aligned}$$



Sidewalk: maintaining widget state

We can save all information needed for a button in a record like

```
data WXButtonState = WXButtonState {
  bsLabel :: String,
  bsEnabled :: Bool
}
```

And then we may pass that information as the *a* parameter in *Widget*. But:

- How can we maintain the current state of the widget?
- How can we declare that we want to *change* something *without* having to *recall* the previous state?

Sidewalk: maintaining widget state

The solution is to use instead the type

$$\textbf{type}\ \mathit{WXButtonConf} = \mathit{WXButtonState} \rightarrow \mathit{WXButtonState}$$

In that way we receive the changes we want to make to the button:

$$\begin{aligned} &\mathit{rec}\ \mathit{count} \leftarrow \mathit{accumulator}\ 0 \prec \mathit{tick} \\ &\quad\ \mathit{tick} \quad \leftarrow \mathit{button}\ \mathit{id} \qquad \prec \mathit{label}\ \mathit{count} \end{aligned}$$

Example: a button

A button also emits clicks, which are *events* that carry no extra info. So the final type for a button widget would be:

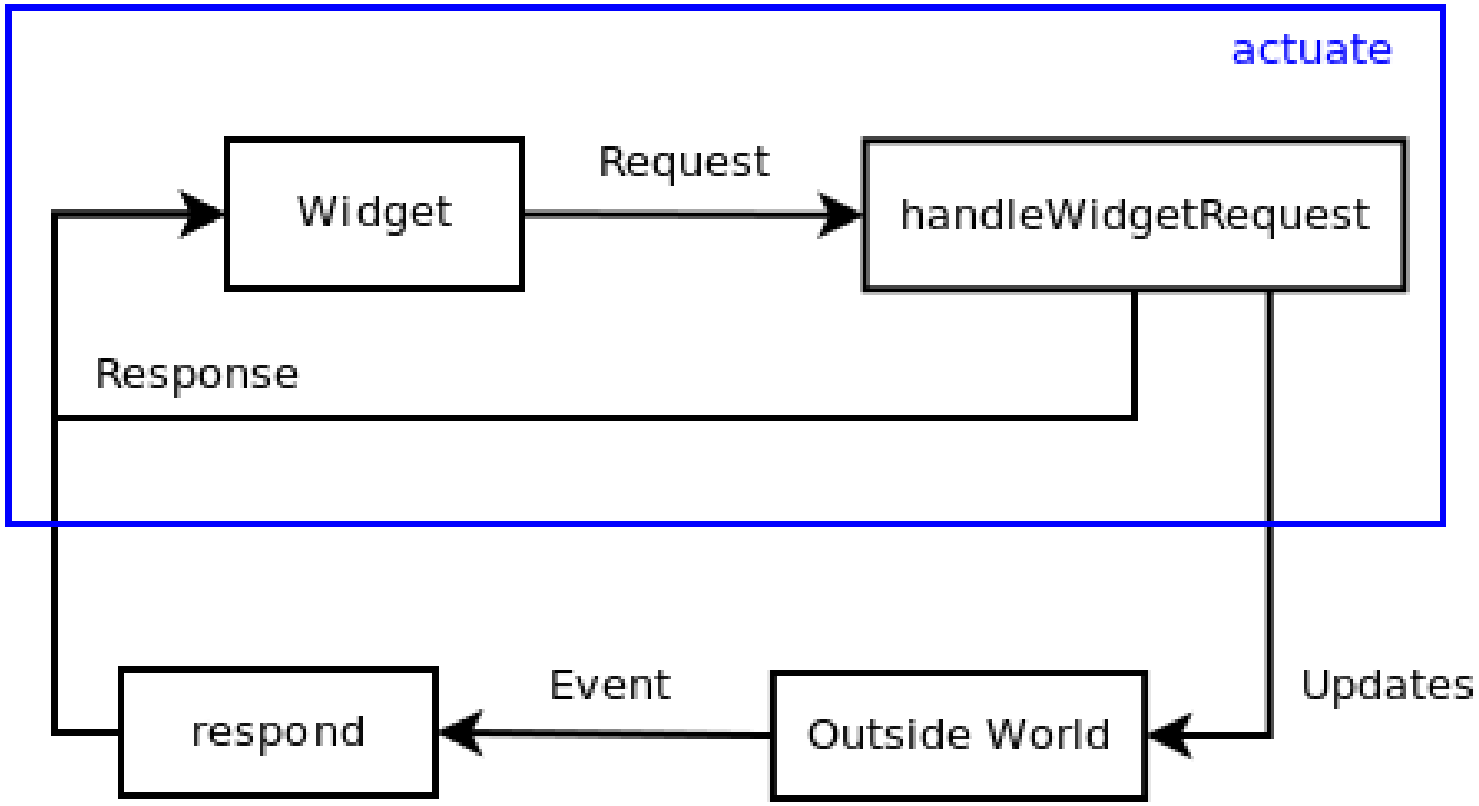
$$wxbutton :: WXButtonConf \rightarrow WXBox\ WXButtonConf\ (Event\ ())$$

Internals

There are three important functions to make this work:

- **actuate** Handles the plumbing of requests and responses.
- **handleWidgetRequest** Listens to messages sent by the widgets, updates the outside world and may generate responses to tell the widgets that the task is completed (e.g. in the initial creation).
- **respond** Function to be called as callback by the outside world, which takes care of "getting up" a widget and sending a new response, maybe starting a new *actuate* loop.

Internals


actuate
Widget
Request
handleWidgetRequest
Response
respond
Event
Outside World
Updates

Layout

Layout is handled by arrow combinators, and wrapping the result into a *wxBox*. Two kind of boxes are provided:

```
wxHBox $ proc _ → do ...    -- wraps horizontally
wxVBox $ proc _ → do ...    -- wraps vertically
```

This wrapping also takes care of bubbling the events to the correct widget.

wxFruit shortcomings

- The set of widgets is fixed in advance
  - Because *WXWidgetReq* and *WXWidgetResp* are hard-coded datatypes
  - You can create new widgets by composing, but it is impossible to wrap new widgets coming from other place without recompiling
  - There was an attempt in 2007 called Juicy Fruit that partially solved this problem

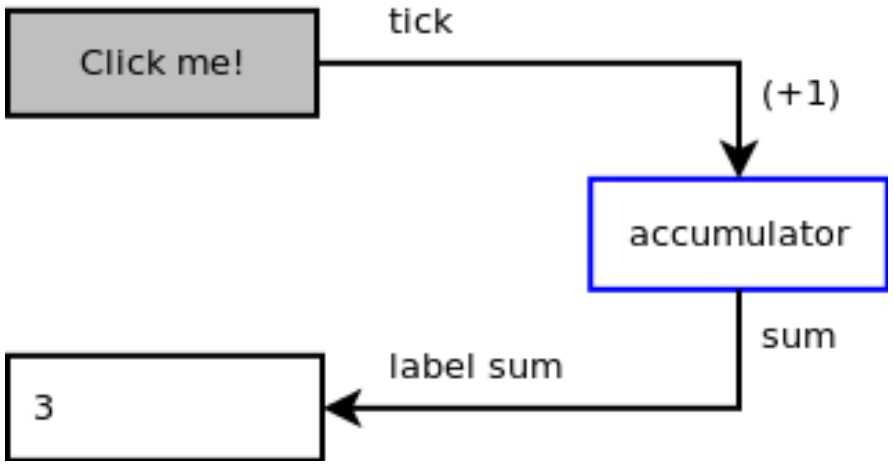
wxFruit shortcomings

- The set of widgets is fixed in advance
  - Because *WXWidgetReq* and *WXWidgetResp* are hard-coded datatypes
  - You can create new widgets by composing, but it is impossible to wrap new widgets coming from other place without recompiling
  - There was an attempt in 2007 called Juicy Fruit that partially solved this problem
- Resource hungry
  - Continuous amount of signals, most of them being *NoEvent*

Our approach

- Use a wxFruit-style of composing interfaces
  - We find the arrow syntax to fit very well
- Don't care about layout, as this is done in HTML+CSS
  - There is a problem when adding widgets, but we don't care at this point either

Intented application to get working



Intented application to get working

$$
\begin{aligned}
&\mathit{accumulator} :: a \to \mathit{SF}\ (\mathit{Event}\ (a \to a))\ a \\
&\mathit{tag} \qquad\qquad :: \mathit{Event}\ a \to b \to \mathit{Event}\ b \\
&\mathit{example} :: \mathit{JSGUI}\ ()\ () \\
&\mathit{example} = \mathit{JSGUI}\ \$\ \mathit{proc}\ _ \to \mathbf{do} \\
&\quad \mathit{tick} \leftarrow \mathit{button}\ (\mathit{id}\ \texttt{"add_one"}) \prec \mathit{id} \\
&\quad \mathit{sum} \leftarrow \mathit{accumulator}\ 0 \qquad\qquad \prec \mathit{tick}\ `\mathit{tag}`\ (+1) \\
&\quad _ \leftarrow \mathit{label}\ (\mathit{id}\ \texttt{"count"}) \qquad \prec \mathit{label}\ \$\ \mathit{show}\ \mathit{sum} \\
&\quad \mathit{returnA} \prec () \\
&\mathit{main} :: \mathit{IO}\ () \\
&\mathit{main} = \mathit{startGUI}\ \mathit{example}
\end{aligned}
$$

Problems we found (1)

UHC currently does not allow Haskell functions as parameters to Javascript functions

- That makes it difficult to register callbacks
- If not, it would be very easy: `$(id).click(f);` (using JQuery)
- We solved it by having a key-value table that was made stateful by saving it in the Javascript partially
  - We could have solved it in a different, more elegant way, but we decided to use this because it was the one with less code that still worked, and it is still viewed as a temporary solution

Problems we found (2)

Bugs in the Javascript backend

- We got a completely compilable version of Yampa just two weeks ago

Problems we found (3)

As said before, the "Fruit approach" generates a massive amount of *NoEvents*

- Needed for have the mouse position correct at every time and, for example, change a label accordingly

Problems we found (3)

As said before, the "Fruit approach" generates a massive amount of *NoEvents*

- Needed for have the mouse position correct at every time and, for example, change a label accordingly

In the Javascript world, by default everything is monothread

- And the dispatcher is not very fair, so at the end nothing was shown on the screen
- Solution: move the event loop to a background thread, using the *JThread* library

Problems we found (3)

As said before, the "Fruit approach" generates a massive amount of *NoEvents*

- Needed for have the mouse position correct at every time and, for example, change a label accordingly

In the Javascript world, by default everything is monothread

- And the dispatcher is not very fair, so at the end nothing was shown on the screen
- Solution: move the event loop to a background thread, using the *JThread* library

Some more problems that may come

- What happens with concurrent upgrades to the DOM?
- Events do not respect order $\Rightarrow$ add event queue?

Problems we found (4)

$$hold :: a \rightarrow SF\ (Event\ a)\ a$$

- This function is used to maintain state in widgets
- But now we have a continuous flood of "set label text" requests for a simple label, which should not be the case
  - This is different from the continuous *NoEvent* flood, which is neccessary
- Basically, this is *the* bug standing in front of getting something that works

Reflection and Conclusion

- We learned about:
  - Utrecht Haskell Compiler
  - ...and its compiler backend
  - Functional Reactive Programming
  - ...specially the Yampa implementation

Reflection and Conclusion

- We learned about:
  - Utrecht Haskell Compiler
  - ...and its compiler backend
  - Functional Reactive Programming
  - ...specially the Yampa implementation
- Future work
  - Make it work!
  - Investigate how to integrate Ajax calls
  - Create some sort of client-side framework, à la Google Web Toolkit

The End

Any questions?