CS4012 Topics in Functional Programming

Glenn Strong

GUI development

As a change of pace, let's look at a library for doing something practical.

Threepenny GUI by Heinrich Apfelmus is a library for creating user interfaces. It is implemented as a ___FRP_ *DSL*.

Threepenny GUI

- Threepenny GUI is based on the earlier "Reactive Banana" FRP library, but simplifies it greatly.
- Creates a browser-based UI
- Unlike, say, Scotty this is intended for use as a *local* web server with a tight interaction loop.

Threepenny GUI

- The library is designed around a weak form of what Gill et. al. called a "Remote Monad".
- Programs expressed in a reactive style by connecting up event flows.
- Haskell code specifies an HTML document. Threepenny converts this to JavaScript that executes on the page.

Example

```haskell
import Graphics.UI.Threepenny

main :: IO ()
main = do
  startGUI defaultConfig showMessage

showMessage :: Window -> UI ()
showMessage window = do
  getBody window #+ [string "Hello, world!"]
  return ()
```

```haskell
startGUI :: Config -> (Window -> UI ()) -> IO ()
```

- `Config` is a value that lets us set up the server (port numbers, etc)
- `Window` represents the DOM `window` object
- `UI` is a transformed IO monad

Reactivity can come from event responses:

```haskell
buttonUI :: Window -> UI ()
buttonUI window = do
  button <- UI.button #+ [string "Click me"]
  getBody window #+ [return button]

  on UI.click button $ \_ -> do
    getBody window #+ [ UI.div #+ [ string "You clicked me!"] ]
```

There is a JavaScript FFI, and JQuery integration:

```haskell
import qualified Graphics.UI.Threepenny as UI
import Graphics.UI.Threepenny.Core

import Graphics.UI.Threepenny.JQuery
```

```haskell
main :: IO ()
main = startGUI defaultConfig setup

setup :: Window -> UI ()
setup w = do
    return w # set title "fadeIn - fadeOut"

    button <- UI.button # set text "Click me to make me fade out and in!"
    getBody w #+ [column [UI.string "Demonstration of jQuery's animate() function"
                         ,element button]]

    on UI.click button $ \_ -> do
        fadeOut button 400 Swing $ do
            runUI w $ fadeIn button 400 Swing $ return ()
```

```haskell
main :: IO ()
main = startGUI defaultConfig setup

setup :: Window -> UI ()
setup w = do
    return w # set title "Mouse"

    out  <- UI.span # set text "Coordinates: "
    wrap <- UI.div #. "wrap"
        # set style [("width","300px"),("height","300px"),("border","solid black 1px")]
        # set (attr "tabindex") "1" -- allow key presses
        #+ [element out]
    getBody w #+ [element wrap]

    on UI.mousemove wrap $ \xy ->
        element out # set text ("Coordinates: " ++ show xy)
    on UI.keydown   wrap $ \c ->
        element out # set text ("Keycode: " ++ show c)
```

There are facilities for graphics (most easily via the canvas), and animation (for example via UI.timer)

State can be managed either: - Explicitly through the IO monad - (demo time) - Using `Reactive.Threepenny`

Threepenny GUI (again)

The same program in threepenny, but using the FRP style

Let's remove the `IORef` from the canvas program

```haskell
import qualified Graphics.UI.Threepenny as UI
import Graphics.UI.Threepenny.Core
import Reactive.Threepenny
```

Setup is the same

```haskell
setup :: Window -> UI ()
setup window = do
  return window # set title "Clickable canvas"

  canvas <- UI.canvas
    # set UI.height canvasSize
    # set UI.width canvasSize
    # set UI.style [("border", "solid black 1px"), ("background", "#eee")]

  fillMode  <- UI.button #+ [string "Fill"]
  emptyMode <- UI.button #+ [string "Hollow"]
  clear     <- UI.button #+ [string "Clear"]

  getBody window #+
    [column [element canvas]
    , element fillMode, element emptyMode, element clear]
```

Now we wire up the reactive event network

First some simple events:

```haskell
let
    efs = const Fill <$ UI.click fillMode
    ehs = const NoFill <$ UI.click emptyMode
    modeEvent = unionWith const efs ehs
```

These are *events* – things that occur at moments in time

`UI.Click` yields a void value, but we can give it a more useful meaning:

```haskell
efs :: Event (Modes -> Modes)
```

We can join two event streams with the `unionWith` operation (if two events fire simultaneously the supplied function will disambiguate)

There are two kinds of interesting `Signal` like things in `Reactive.Threepenny`

- Events (things that happen at particular moments in time)
- Behaviors (continuous things that have values)

We can go from events to behaviors using the `stepper` function:

```
mousePos <- stepper (0,0) $ UI.mousemove canvas
```

```
stepper :: MonadIO m => a -> Event a -> m (Behavior a)
mousePos :: Behavior (Int,Int)
```

Another way to make the `Event` to `Behavior` leap is with the `accumB` helper function:

```
drawingMode <- accumB Fill modeEvent
```

```
accumB :: MonadIO m => a -> Event (a -> a) -> m (Behavior a)
drawingMode :: Behavior Modes
```

We want to combine our two `Behavior` values, and use the result when it's time to draw.

```
let bst = (,) <$> drawingMode <*> mousePos
```

Here we are making use of the fact that `Behavior` is an `Applicative`

```
bst :: Behavior (Modes, (Int,Int))
```

This relies on the expected identify for applicatives:

```
do
  x <- f1
  y <- f2
  return f0 x y
```

should be the same as

```
f0 <$> f1 <*> f2
```

We want the behaviour available to examine in our drawing event, so we really write:

```
let bst = (,) <$> drawingMode <*> mousePos
    eDraw = bst <@ UI.click canvas
```

To finish up:

```
onEvent eDraw $ \e -> do drawShape e canvas

on UI.click clear $ const $
  canvas # UI.clearCanvas
```

A helper function for drawing:

```
drawShape :: (Modes, (Int,Int)) -> Element -> UI ()
drawShape (Fill, (x,y)) canvas = do
  canvas # set' UI.fillStyle   (UI.htmlColor "black")
  canvas # UI.fillRect (fromIntegral x,fromIntegral y) 100 100
drawShape (NoFill, (x,y)) canvas = do
  canvas # set' UI.fillStyle   (UI.htmlColor "white")
  canvas # UI.fillRect (fromIntegral x,fromIntegral y) 100 100
```

Arrows and FRP

To discuss FRP in more detail we'll start by looking at Arrows, which give us an abstract introduction.

- Arrows are a relatively new abstraction for function-like types
- More general than monads (every monad introduces an arrow but not vice-versa)

- Monads allow us to combine actions in a linear way
- Arrows allow some more interesting combinations
- They seem particularly well suited to stream processing
  - Right now the most common applications are in FRP (games, music, and other reactive systems)
- Originally defined by John Hughes (some of the following examples come from his paper)

- Like Monads Arrows are represented in Haskell by a type class
  - (the actual definition is a little more involved than this…)

```
class Arrow a where
    arr   :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
```

Compare this to the `Monad` class

```
class Monad a where
    return :: b -> a b
    (>>)   :: a b -> a c -> a c
```

We can think of monads as wrapping their results in containers; arrows, by contrast, wrap the operations themselves in containers.

- Motivation?
- Here's a Haskell function that counts word matches

```
count :: String -> String -> Int
  count w = length . filter (==w) . words
```

To compose this with something that does IO we have to do lifting

```haskell
countFile :: String -> FilePath -> IO ()
countFile w = (>>= print) . liftM (count w) . readFile
```

Lifting?

```haskell
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = do { x <- m; return (f x) }
```

The Arrow typical lets us rewrite this in terms of combinators that we can apply to both normal functions and monadic functions

```haskell
countFile :: String -> Kleisli IO FilePath ()
countFile w = Kleisli readFile
              >>> arr (count w)
              >>> Kleisli print
```

`arr` is like liftM but for Arrows:

```haskell
arr :: Arrow a => (b -> c) -> a b c
```

`Kleisli` makes arrows out of Monadic functions

How does this relate to reactive programming?

- A typical small example is the arrow of stream functions

```haskell
newtype SF a b = SF { runSF :: [a] -> [b] }
```

Which we might instantiate like this:

```haskell
instance Arrow SF where
    arr f = SF (map f)
    SF f >>> SF g = SF (f >>> g)
```

So we can apply this across streams of values:

```haskell
runSF (arr (+1)) [1..5]
```

- We need many more operations to have full computational power
- These are divided across several classes (as not all arrows can meaningfully define them all).
  - Arrows that support choice (the `ArrowChoice` class)
  - Arrows that support application (`ArrowApply`)
  - Arrows that support feedback (`ArrowLoop`)

Because of the theoretical underpinnings of arrows we actually derive them from the class Category

```haskell
class Category cat where
    id  :: cat a a
    (.) :: cat b c -> cat a b -> cat a c
```

From this we have the chaining functions:

```
(>>>) :: Category cat => cat a b -> cat b c -> cat a c
(<<<) :: Category cat => cat b c -> cat a b -> cat a c
```

The basic Arrow class is then declared as:

```
class Category a => Arrow a where
    arr ::(b->c)->a b c
    first  :: a b c -> a (b, d) (c, d)
    second :: a b c -> a (d, b) (d, c)
```

first and second make a new arrow by transforming the first (or second) argument of an existing arrow.

```
(***) :: a b c -> a b' c' -> a (b, b') (c, c')
(&&&) :: a b c -> a b c' -> a b (c, c')
```

These functions combine arrows by:

- `(***)`: running two arrows on a pair of values (one arrow on the first, the other on the second)
- `(&&&)`: running two arrows on the same value