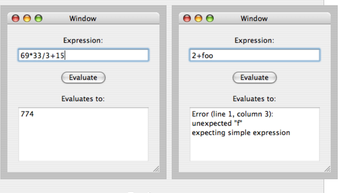
HOC: A Haskell to Objective-C Binding

Adding a GUI to an Expression Parser

You've probably seen many examples of Haskell GUI libraries which show you how to create a simple graphical application that has a cute little button which prints "Hello, world!" when you click on it. Well, we'd show you that, but the truth is it's so easy to do with Cocoa that it'd just be boring (as well as being useless). So, let's do something a bit more practical with our first example: our mission is to put a simple GUI around a basic Haskell program, which is something you're more likely to be interested in.

For this quick tour, let's write a GUI program that allows a user to type in arithmetic expressions (e.g. `2+5` or `69*33/3+15`), and print its result. Something like this will do:



Preface

In comparison to other GUI toolkits available for Haskell, there's a bit more of a learning curve to use HOC for writing GUIs, because Cocoa enforces good design by separating your application logic cleanly using design patterns such as Model-View-Controller (which we will use in this example). So, while this quick tour may seem a bit longer than you expect, the really good news is that Cocoa's design patterns *scale* very well: writing a small GUI may take 50-100 lines of code, but writing a much bigger, fancy GUI with lots of widgets and controls may only take 100-200 lines rather than 500-1000, because much of the GUI code is handled for you by Interface Builder's target-action/outlet design. In Mac OS X Panther, you can even use Cocoa Bindings with your Haskell application, and cut down your code size even further.

Note that HOC is also a lower-level binding than higher-level GUI toolkits such as Fudgets or even wxHaskell: it is really a bridge which enables you to use Objective-C objects from Haskell (and thus gives you much more functionality than simply being able to write GUIs!). In the future, we hope to layer a higher-level interface on top of HOC to provide a more functional API, so you're not forced into imperative style of coding as you would write Objective-C code. Even so, the Cocoa framework is so well designed that the resulting imperative code is still far shorter than it would be compared to most other frameworks .

The Model: an Arithmetic Expression Parser

Our first step is to write the arithmetic expression parser. To do this, we could use Daan Leijen's most excellent Parsec parser combinator library. In fact, Parsec comes with such excellent documentation that it even gives example code which shows how to do this. So, imagine the following code is in an `ExpressionParser.hs` Haskell module:

```
module ExpressionParser
where

import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr

expr :: Parser Integer
expr = buildExpressionParser table factor <?> "expression"

table = [ [op "*" (*) AssocLeft, op "/" div AssocLeft]
        , [op "+" (+) AssocLeft, op "-" (-) AssocLeft] ]
  where op s f assoc = Infix (do { string s; return f }) assoc

factor = do { char '('; x <- expr; char ')'; return x }
         <|> number
         <?> "simple expression"

number :: Parser Integer
number = do { ds <- many1 digit; return (read ds) } <?> "number"
```

There's probably around 10 lines of code there, but if you run the code in GHCi, it seems to work quite well:

```
*ExpressionParser> parseTest expr "2+5*2"
12
*ExpressionParser> parseTest expr "69/3+1"
24
```

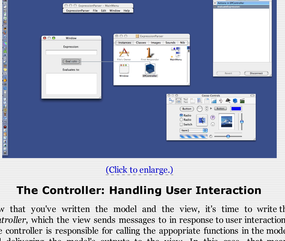
So, you have a working expression parser. Unfortunately, you have to to run this from GHCi to use it. You could write a command-line interface to it, but wouldn't it be nicer if you could run it from a Haskell GUI instead?

The View: a GUI for the Expression Parser

Now that you've written the model, it's time to write the *view*, which lets the user interact with the model by clicking on buttons, selecting menu items, and all that fancy GUI stuff. Luckily, the Cocoa framework on Mac OS X makes this quite trivial: instead of writing tedious layout code to draw the view, set up the menus, etc., we use Interface Builder, a GUI design tool. Briefly, the steps involved with Interface Builder are:

- Drag the text fields and buttons from the Cocoa Controls palette into your window.
- Tell Interface Builder that you'll be writing a class called `EPController` to handle events such as the user clicking on buttons. To do this, you simply subclass and instantiate the `NSObject` class (the root of all of Cocoa's classes) in Interface Builder's Classes window.
- Add two *outlets* to the `EPController` class, which are pointers to the two text fields in the window. This enables your controller class to read the user's typed-in expression from the first text field, and write its output to the second text field.
- Add an *action* to the `EPController` class, which contains the expression-evaluation code that will run when the user clicks on the "Evaluate" button (or presses the *Enter* key in the input text field).
- *Connect* the text fields to `EPController`'s outlets by Ctrl-dragging from the instantiated `EPController` object to the text field, and setting the outlet appropriately. You also want to connect the Evaluate button and text field entry to `EPController`'s, again by Ctrl-dragging from them to the instantiated `EPController` object icon.

That's it: your view's done. You could write all this code programmatically of course, but why bother? Interface Builder allows you to add GUIs to your code very rapidly, and also handles issues such as window re-sizing elegantly. (This is often quoted as a reason to manually write code—so you can use a geometry manager—rather than using a GUI designer tool. Truth is, Cocoa and Interface Builder's re-sizing controls are so easy and powerful that you'll very rarely have to write any geometry management code—I've *never* had to, even when writing a full-fledged media player application!) Here's a screenshot of what Interface Builder will look like as you're designing your view with it:



(Click to enlarge.)

The Controller: Handling User Interaction

Now that you've written the model and the view, it's time to write the *controller*, which the view sends messages to in response to user interactions. The controller is responsible for calling the appropriate functions in the model, and delivering the model's outputs to the view. In this case, that means delivering the expression the user typed in to the `ExpressionParser` module's parsing code, taking the parser's output, and displaying that back in the view.

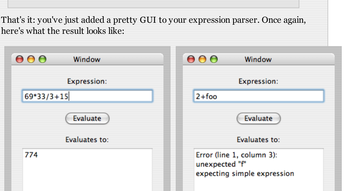
HOC enables you to write the controller's code in pure Haskell: first, you need to write a `Selectors.hs` file which declares all the *selectors* (method names) that you will be using:

```
{-# OPTIONS -fglasgow-exts #-}
module Selectors where

import AppKit.NSButton

$(declareSelector "evaluateExpression:" [t| forall a. NSButton a -> IO () |])
```

This enables better static checking of method names, and prevents scenarios such as pondering why your method wasn't being called when you actually misspelled it (which can happen with Objective-C). After this, it's time to write the controller code:

```
{-# OPTIONS -fglasgow-exts #-}

module EPController where

import Cocoa hiding (parse)
import ExpressionParser
import Selectors
import Text.ParserCombinators.Parsec (parse)

$(declareClass "EPController" "NSObject")

$(exportClass "EPController" "ep_"
  [ Outlet "expressionEntry" [t| NSTextField () |]
  , Outlet "evaluation"      [t| NSTextField () |]
  , InstanceMethod Selectors.info_evaluateExpression ])

obj #. var = obj # getIVar var

ep_evaluateExpression _self = do
  -- Get the expressionEntry outlet text field from this object, and get
  -- what the user typed as a Haskell string
  expression <- self #. _expressionEntry >>= stringValue >>= haskellString
  -- Parse the expression
  case (parse expr "" expression) of
    Left err ->
      -- Parsing returned an error: display it in the output text field
      self #. _evaluation >>= setStringValue (toNSString $ "Error " ++ show err)
    Right answer ->
      -- Parsing was successful: display the answer
      self #. _evaluation >>= setStringValue (toNSString $ show answer)
```

That's it: you've just added a pretty GUI to your expression parser. Once again, here's what the result looks like:



Of course, this is only a quick overview on how to use HOC: this brief look at HOC is expanded upon in the HOC documentation, where we describe the exact steps involved in building this arithmetic expression parser. We hope that this example gives you an idea of how you can use HOC to easily implement a GUI for your existing Haskell applications!

SOURCEFORGE.NET®