Lennart Jansson

CS240H Final Project, June 11 2014

Shiny: A Framework for Client-side Web Apps

Motivation and introduction

Anyone who has tried to build a web application that runs in a browser with only raw Javascript and with no overarching framework will have realized that trying to build an app this way is downright awful! Plain DOM manipulation is very tedious, and even a simple web app can quickly grow into spaghetti when DOM event handlers bound to elements on the page are mixed with code to generate new elements or modify content or styling or perform logical calculations needed for the app to function. Such web apps are extremely fragile and small changes to logic can completely break functionality of the tightly coupled event handlers.

The solution is, of course, to enforce some sort of design structure on the web application. The most popular pattern is MVC (model-view-controller), which separates out the concerns of code that makes sure the app renders properly in the browser (the view), the underlying data structures (models) and the controller that ties the two together with the logic of the application. There is such a prevalent need for Javascript MVC frameworks, but also such a variety in interpretations of the precise semantics of MVC, that there are now hundreds of competing libraries that attempt to enforce structure on an app and manage boilerplate related to DOM manipulation (see http://todomvc.com/).

Unfortunately, Javscript MVC frameworks still require a developer to write one's code in Javascript, a relatively poor language that contains robustness issues, strange typing, and syntactic gotchas. What if one could use a pure strongly-typed language instead to write web apps? The developer's life would be easier, since the language could provide guarantees about what is pure and effectful for cleaner organization of code.

This is essentially what I've aimed to do here with Shiny, a Haskell library that can provide similar functionality to any other Javascript MVC framework, such as clean data-binding of internal app data to what the users sees in the browser page. Shiny can produce Haskell applications that can be compiled to JS using the magical compiler GHCJS (https://github.com/ghcjs/ghcjs), which is an alternate backend to the GHC compiler that allows

STG code to be translated into Javascript. Among other things, GHCJS features a complete port of the GHC runtime system to Javascript, so any possible Haskell program that relies on advanced GHC features (all language extensions) and the properties of the runtime (GC, scheduling, MVars) can still be compiled to Javascript and run in any modern browser. Conveniently, nearly every package on Hackage can work with a GHCJS program (the only exception is libraries that require external native code).

Shiny is extremely WIP, and doesn't actually go far enough in power to support all reasonable use cases. I'll describe its limitations later. But I'll present an initial demonstration of how internal application state (the model) can be conveniently bound to the browser view in a way that separates concerns and preserves modularity, all with a very clean syntax.

Example use and design

I haven't given snippets of code here; please view the Main.hs file in the Shiny source distribution I've attached, which presents a simple example proof-of-concept web application. The design considerations I describe here can be easily seen in this example app. A Shiny app has type `Shiny`. We can run an app as the main action of our program (when compiled to JS using GHCJS) by running `main = runShiny (myShinyApp :: Shiny)`. Basic static HTML elements have type `Shiny`, and can be styled with a syntax inspired by BlazeHtml (http://jaspervdj.be/blaze/).

`Box`es store the application state that may be changed while the app is running. Since they essentially contain mutable variables, they must be created in the IO monad. So all boxes are created when the app is initialized, and then the Shiny app can be defined to contain references to the created boxes.

The goal with the design of Shiny's syntax is to clearly show an AST of the DOM as in BlazeHtml, with the syntax for binding the content of boxes to special "shiny" elements. For example, we have a `shinyLabel` that always shows the current context of a box, and changes its content when the box changes. We might then want an HTML input element to be able to change the value of the box—this can be accomplished with a `shinyTextInput`, which writes its value to a box whenever the HTML input's content is changed.

Implementation details

Shiny uses the ghcjs-dom library to perform its internal DOM manipulation. Nearly all raw DOM manipulation, however, is hidden from an application developer, who can deal only in the Shiny primitives and static HTML elements.

Shiny chooses to represent the state of the application in two ways. The first is through boxes, which contain the inner application state that is exposed through bindings and the view. This is the state that the user application can access and modify. The second kind of state is the current state of the DOM altogether. The Shiny data type internally represents a whole DOM tree at any one instant, as well as the information necessary for model->view bindings. A data binding on a Shiny element allows modification of any of the content of its associated DOM node and children nodes. Since DOM node attributes and content (and really the whole tree structure) could change at any time due to the properties of a particular data binding, the internal Shiny type that reflects the DOM must also be able to change as a hidden state variable.

Instead of dealing with the complexities of many local states variables, one for each node, we choose the much simpler implementation of storing exactly one global Shiny that represents the current global state of the DOM. But then a question arises: how can we let Shiny elements modify themselves while wrapped up in the context of a monolithic global variable? The answer is, of course, to apply copious quantities of Edward Kmett's lenses. The whole Shiny tree is subjected to preprocessing when the app is initialized that endows each Shiny node in the tree with a lens on the global Shiny to the localized Shiny at the current point of traversal. That way, when the Shiny element wants to launch a thread (this happens during initialization) that will update itself in response to changing content, as long as this modifier has a reference to both the mutable global Shiny and the lens to the sub-Shiny, the subroutine can modify the global data type over the lens and thus perform the correct operation.

Lenses provide a method for encapsulation so that Shiny elements cannot break the whole DOM tree willy-nilly, while still giving us the benefits of a single global state, such as a simplicity in reasoning about state transitions and guaranteed coherency. One downsides is possible inefficiency: whenever a Shiny node needs to update, thankfully we don't have to regenerate and re-render the entire DOM tree from its document root, but we do still need to regenerate the subtree of the DOM that corresponds to the modified element. If this element has

many children or is near the root, then the cost is almost as bad. Whenever a Shiny updates, we must do one additional traversal over all of its arbitrary depth children in order to update DOM pointers and associated lenses, so that these do not go out of date.

The boxes which provide the mutable application state abstraction have a relatively simply implementation. Each box contains an MVar to the current value and a Chan of updating values. Whenever a Shiny element needs to modify a box, the the new value of the box is written into its Chan. For a thread to listen to changes to the value of a box, it supplies a callback which will be run with the new value, and a new thread is forked which duplicates the box's Chan, then loops forever, waiting for values to come through the duplicated Chan. This is a very simplistic implementation of a publish/subscribe system, and may not be the most efficient that can be written for GHCJS. But it was easy to implement and it does the job.

The ties (data binding) are implemented with the existential types GHC extension. A Shiny element way need to be bound to an arbitrary number of different data sources, each with their own Setter into the Shiny for performing a mutation on the Shiny element. Since the data types in the boxes might really be anything, we need the Shiny to store a heterogenous list of bindings. The solution is the interesting type `data Tie = forall a. Tie (Box a, Setter' Shiny a)` of which the Shiny might store a list. When dealing with existential types, we have to be careful not to let the hidden `a` type parameter leak to the outside. However, a whole data binding can be accomplished without prior knowledge of what the `a` will be: on the box, set up a listener with a callback that will take the unknown `a`, but then use the `Setter' Shiny a` to modify the global Shiny. The fact that we always carry around a `Setter'` of the correct type makes it possible to heterogenously bind a single Shiny to many data sources without leaking type parameters.

Further work

If you view the example code, you'll find the Shiny library in its current state does have something resembling a "foreach" binding. This is an essential binding for any reasonably complex application—for example, it's absolutely necessary for a todo-list app, so that there can be a variable number of todos in the list and they all render one after the other. The application developer would wish to be able to specify how a single todo item would render, and they apply

this subview to the foreach binding to generate the full view of the todo list. The Shiny library currently does not fully support this. While it is possible to generate a subview for a todo item and use the foreach binding to generate a list of items, it is not possible to have the subitems independently interactive. This would require new mutable boxes for each item, and in my library the boxes are insufficiently flexible.

One possible improvement would be to make boxes an instance of Functor and Applicative, so that when a certain box updates, any derived boxes would also update, and the Applicative instance would allow combining together the data streams from many boxes into one. With clever combinators, this might solve the flexibility problem that prevents a todo list from being implemented.

But why stop there? A `Box a` is really just a bastardized `Signal a` from any reasonable formulation of FRP (functionally reactive programming). FRP may be the result of taking the ideas from the implementation of Shiny to their logical conclusions in search of greatest elegance. Is it practical to write web apps in GHCJS with FRP? Maybe not right this instant, but it should be very soon! https://github.com/ghcjs/ghcjs-sodium is a Google Summer of Code project to create convenient HTML manipulation bindings for the Sodium FRP Haskell library. If this is successful, it may be possible soon to easily build web apps using FRP, a very clean theoretical idea. Shiny is an interesting experiment, but it reveals limitations with simply plunging in and implementing a data-binding system for MVC-like patterns without a good theoretical foundation. Much more research is required as to the best and most elegant method for developing web applications, though working with GHCJS and Haskell is a great way to start.