Haskell DSLs for Interactive Web Services

Andrew Farmer   Andy Gill

Information and Telecommunication Technology Center
The University of Kansas
{afarmer,andygill}@ittc.ku.edu

Abstract

Robust cross-platform GUI-based applications are challenging to write in Haskell, not because providing hooks into existing GUI frameworks is hard, but because these hooks are both platform dependent and prone to rapid bit-rot. Browser-based user interfaces implemented using Javascript and HTML5 offer some relief from this situation, providing a standardized cross-platform API. However, Javascript is a client-side scripting language, and a traditional shallow, foreign-function-call style Haskell interface to the client does not scale well because calling a Javascript command involves sending the command, and optionally waiting for a response, over a network. Instead, we build a deep embedding of Javascript commands inside a Javascript monad. Along with supporting web-based infrastructure, we deliver entire program fragments to our web-based applications. Using our monad, the Haskell programmer can choose the granularity of interaction, with the option of having reactive interfaces that are completely handled by the client.

1. Introduction

Haskell support for GUI libraries is a long and sad story with many interesting actors. The reasons for this – in part – is the cross-model problem. That is, well supported GUI libraries are written in other languages, use other languages' idioms, and tend to be platform or deployment mechanism specific. Pure Haskell frameworks are not mature or comprehensive enough for large applications. Cross-model frameworks, such as wxHaskell, leverage other people's efforts, but quickly bit-rot, being dependent on a binary interface to a specific version of the underlying library. If one is prepared to commit to one platform, the situation is manageable, but cross platform solutions often involve finding command-line options for specific versions of sub-packages that are not written in Haskell. The challenge of compiling a working ThreadScope [8] on OSX Lion is formidable, even for seasoned Haskell programmers.

Web-based applications have become increasingly sophisticated over the last few years. Facebook is a web application used by almost a billion people. Google Maps interactively allows zooming and exploration. Google Documents provides a useable set of office tools for casual use. So the question we are asking is: can we use Haskell and some cross-model techniques to subsume the Haskell GUI graveyard problem?

This paper introduces `sunroof`, our custom DSL for writing interactive web-based applications, and two other Haskell DSLs that help connect Javascript in the browser to Haskell on a server.

- `sunroof` is a Javascript compiler, compiling monadic combinators in Haskell to Javascript that is executed on the browser. (§4)
- `kansas-comet` provides the ability to transparently query and manipulate the browser from the server, using Javascript. (§3)
- `scotty` provides a declarative DSL for dispatching incoming RESTful [6] web requests to appropriate Haskell code. (§2)

These libraries all sit on top of the popular Haskell web server `warp` [13]. All three DSLs make critical use of functional programming ideas to deliver their functionality. We now discuss our DSLs from bottom to top.

2. Scotty

The popular Warp HTTP server handles the heavy lifting of managing connections and protocols, accepting a callback function that implements application-specific logic with the type:

```
Request -> <warp-monad> Response
```

Scotty is a lightweight web framework that generates this application callback. The programmer specifies *route patterns* which map to corresponding Haskell code that builds a response, which is returned to the client. This design supports the creation of RESTful web services, which provide a uniform interface to server resources via URIs and HTTP verbs and response codes. As an example:

```
fib :: Int -> Int
...

main = scotty 3000 $ do
    get "/fib/:n" $ do
        n <- param "n"
        text $ pack $ show $ fib n
```

Running this program starts the Warp server listening on port 3000 and responds to GET requests for the URI `/fib/`*n*, where *n* can be any integer. The *route pattern* contains a named *capture*, denoted by a colon, named `n`. The response body will be the result of calling `fib` on the value of the capture, which is returned by `param`.

Scotty's design is taken directly from that of a popular Ruby web framework called Sinatra. Indeed, we named it Scotty because it represents the combination of Sinatra and Warp. We leverage Haskell's type system to provide smarter routes than Sinatra. In the above example, `n` is an `Int`, and attempts to parse any ill-formed URI (for example "`/fib/foo`") will fail to match. In this way, the programmer can always count on input being correctly typed in the body of the route, overlapping routes are differentiated by the type of the captures, and Scotty avoids some of the ad-hoc input checking often needed in Ruby's Sinatra. Scotty is further documented on Hackage [5].

3. Kansas Comet

Comet is a web application design pattern where the server is able to push data to the client [10]. Kansas Comet provides generic support for the Comet idiom, on top of Scotty.

There are several techniques for implementing Comet, broadly divided into streaming solutions, where an open connection is maintained for the life of the application, and long polling solutions, where the client periodically checks for new data. At the moment, streaming solutions are finicky and notoriously browser-specific, so Kansas Comet makes use of long polling.

The concept is simple – each Kansas Comet webpage has a software transactional memory variable (TMVar) that acts as a mailbox. Server processes may place Javascript code in the TMVar. The client regularly requests the contents of this mailbox, and runs any code in the response. If the server has nothing to be done, it holds the incoming request, and waits until it does have something (or a Haskell timeout happens). A TMVar is used rather than a channel to make sure that the client and server stay in sync.

The client side of Kansas Comet is provided as a jQuery plugin. When the page loads, a simple connect function starts the polling process. When a connect request arrives at the server, a unique session identifier is generated and returned to the client. This identifier is passed by the client on all subsequent requests, and is used to keep track of the TMVar mailboxes.

The server side of Kansas Comet is presented as a Scotty application, which can be embedded in existing Scotty applications, or hosted directly. The programmer provides a callback to Kansas Comet which implements the application logic. As an example, here is a Comet application which causes a browser alert whenever the user clicks on a tag which is a member of the 'click' class.

```
web_app :: Document -> IO ()
web_app doc = do
    register doc "click" "return {};"
    forever $ do
        res <- waitFor doc "click"
        send doc "alert('clicked!');"
```

The call to register tells the client to register an event listener that returns an empty object when the mouse is clicked. Then waitFor blocks until a click event is posted by the client. Kansas Comet also has direct support for queries (not shown), where the result of a Javascript computation can be sent back to Haskell, and Haskell will wait for the result.

The roundtrip required for events and queries can be a serious performance bottleneck. In a drawing application that acts on every *mouse movement*, the rate of event generation will be high, and it is preferable that the client does the work as much as possible. To that end, we wish to compile our reactive application logic into Javascript for direct execution in the browser.

4. Sunroof

Javascript implementations on all major browsers provide a powerful API for building interactive web pages. Libraries like jQuery and jQueryUI build on this, providing a browser-independent set of graphical widgets and other capabilities. We want to use these libraries, but program in Haskell.

A usable model can be built using a simple translation of a fixed set of function calls into Javascript commands. With careful construction, we can combine commands before sending them, optimizing network usage. The challenging part is having the Javascript return values in an efficient manner. Consider this Haskell code:

```
c <- getContext "my-canvas"
c <$> beginPath()
c <$> arc(x, y, 20, 0, 2 * pi, false)
c <$> fillStyle := "#8ED6FF"
c <$> fill()
```

In a simple transaction model, getContext invokes a Javascript command on the client, returning the response as c, a handle to a canvas element, which we then invoke Javascript commands on. However, there is no need to perform a round trip to the server, and we would prefer the whole code fragment to be compiled to Javascript such that the binding and use of c are performed on the client directly, with no intermediate client $\leftrightarrow$ server communication.

What we want to do is reify our Javascript monadic program, using the techniques from Elliott et al. [4], which compiled Haskell expressions by providing a deep embedding of overloaded arithmetical syntax, and other operators. In order to allow monadic reification, we need to require a class constraint on the monadic bind so that we can generate a prototypical version of the value passed through bind. Unfortunately, we cannot add post-hoc constraints on existing classes.

We can, however, simulate this behavior by adding constraints to all our primitives. The folklore has been that extracting a monadic deep embedding with polymorphic primitives was not possible. Surprisingly, this turns out to be a straightforward application of existing techniques for unrolling monads and using standard Haskell overloading to provide prototypical values.

To compile our Javascript monad, we use the Hackage package operational [1], which builds on the ideas found in Unimo [9]. This package uses the left identity and associativity monad laws to normalize a monadic program into a stream of primitive instructions terminated by a return.

$$\begin{array}{lcl} Program & ::= & Primitive \texttt{ >>= } Program \\ & | & \texttt{return}\ \alpha \end{array}$$

We then constrain the returned values of all the primitives to be reifiable via a constraint on GADT constructors. In (the simplified version of) our compiler, Javascript function calls are implemented with the JS_Call primitive.

```
data JSInst a where
  JS_Call     :: (Sunroof a)
              => String -> [JSValue] -> JSInst a
  ...
```

From this list of primitives, the operational package allows us to build our Javascript monad, with the monad instance for JSM is provided by Program.

```
type JSM a = Program JSInst a
```

For technical reasons, Program is abstract in operational, so the library provides the view function to give a normalized form of the monadic code. In the case of JS_Call, bind corresponds to normal sequencing, where the result of the function call is assigned to a variable, whose name has already been passed to the rest of the computation for compilation. The Sunroof class provides newVar, assignVar, and showVar.

```
compile :: Sunroof c => JSM c -> CompM String
compile = eval . view
  where
    showArgs :: [JSValue] -> String
    showArgs = intercalate "," . map show

    eval :: Sunroof b
         => ProgramView JSInst b -> CompM String
    eval (JS_Call nm args :>>= g) = do
       a <- newVar
       code <- compile (g a)
       return $ assignVar a ++ nm ++ "("
                ++ showArgs args ++ ");" ++ code
    ...
    eval (Return b) = return $ showVar b
```

Our compiler critically depends on the type-checking extensions used for compiling GADTs, and scales to additional primitives, provided they are constrained on their polymorphic result, such as JS_Call.

Using this function, we compile our sunroof Javascript DSL to Javascript, and now a bind in Haskell results in a value binding in

Javascript. The send command compiles the Javascript expressed in monadic form and sends it to the browser for execution.

```
send :: (Sunroof a) => JSM a -> IO a
```

The Javascript code then responds with the return value, which can be used as an argument to future calls to send. Finally, send is thread-safe (two Javascript scripts can run concurrently in the same session), and scripts that return unit are (by design) asynchronous.

We can write a trivial example which draws a circle that follows the mouse:

```
drawing_app :: Document -> IO ()
drawing_app doc = do
  ...
  send doc $ loop $ do
        event <- waitFor "mousemove"
        let (x,y) = (event ! "x",event ! "y")
        c <- getContext "my-canvas"
        c <$> beginPath()
        c <$> arc(x, y, 20, 0, 2 * pi, false)
        c <$> fillStyle := "#8ED6FF"
        c <$> fill()
```

The following code is generated by Sunroof (on the Haskell server) and then executed entirely on the client:

```
var loop0 = function(){
  waitFor("mousemove",function(v1){
    var v2=getContext("my-canvas");
    (v2).beginPath();
    (v2).arc(v1["x"],v1["y"],20,0,2*Math.PI,false);
    (v2).fillStyle = "#8ED6FF";
    (v2).fill();
    loop0();
  })
}; loop0();
```

Each Javascript function used from Haskell is declared using a simple wrapper around the raw constructors. For example, getContext is defined using:

```
getContext :: JSString -> JSM JSObject
getContext nm = JS_Call "getContext" [cast nm]
```

Functions like getContext can be provided in libraries on top of sunroof, and these libraries can be used to implement the interactivity. We expect that actual amount of JSM code to be small in a large application. Finally, we have glossed over a number of reification issues that we have covered in previous publications, especially how we handle embedded conditionals [7]; a known challenge when using deeply embedded DSLs. Work continues to place new DSLs on top of sunroof, and explore various possible trade-offs of our Javascript monadic language.

5. Related Work

There are numerous Haskell web frameworks, including HAppS, Snap, and Yesod. Scotty addresses a much narrower slice of the stack than these more full-featured frameworks. Each of them could be used to implement a Comet server, but, to the best of our knowledge, no ready-made plugin exists.

The UHC compiler offers a backend [14] that compiles Haskell to Javascript which has been used to implement a web application [15]. The GHCJS project [11] is an experimental attempt to add a Javascript backend to GHC. Additionally, there are a number of Haskell-hosted Javascript-generating DSLs, such as JMacro [3] which works via quasiquotation and supports hygenic name generation. The Yesod framework has an experimental means of generating jQuery code [12].

Using a data-structure to represent a deep embedding of a monad has been used before, including Unimo [9] and operational [1]. A monomorphic deep embedding of a monad has been previously reified using operational by Apfelmus [2]. We believe that our observation about using class constraints on the primitives for a monadic language has increased the expressiveness of monadic programs that can be realistically reified.

The precursor to sunroof was our blank-canvas DSL, where monadic primitives were statically constructed to not return values, and interactions were initiated by listening to specific events, meaning every interaction was handled on the server side. Even with this significant performance restriction, the package has been used by several students to write basic interactive games in Haskell, including an animated version of Connect Four, which gives us hope that sunroof will also prove popular with our students.

6. Conclusion

By providing an executable specification language for RESTful web services, a push-based session manager, and a Javascript monad compiler, straightforward cross-platform graphical interfaces can be provided in Haskell. All three utilize functional programming tricks, including catching type-coercion failures in Scotty, software transactional memory in Kansas Comet, and the novel monadic embedded DSL in Sunroof. The compilation of monadic code in particular opens up the possibility of cross-compilation in specific, targeted applications, such as web programming, without needing a complete Haskell to Javascript compiler, and all the accompanying challenges that go with it.

7. Acknowledgments

We would like to thank Heinrich Apfelmus for pointing out that we could rework our compiler to use operational, and providing us with a suitable template of its usage.

References

[1] H. Apfelmus. http://hackage.haskell.org/package/operational.

[2] H. Apfelmus. The operational monad tutorial. *The Monad.Reader*, (15):37–55, January 2010.

[3] G. Bazerman. JMacro. http://www.haskell.org/haskellwiki/Jmacro.

[4] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[5] A. Farmer. http://hackage.haskell.org/package/scotty.

[6] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[7] A. Gill and G. Kimmell. Capturing functions and catching satellites. ITTC Technical Report ITTC-FY2011-TR-29952011-1, Jan. 2011.

[8] D. Jones, Jr., S. Marlow, and S. Singh. Parallel performance tuning for haskell. In *Haskell Symposium*, pages 81–92, 2009.

[9] C.-K. Lin. Programming monads operationally with unimo. In *ICFP*, pages 274–285, 2006.

[10] M. Mahemoff. HTTP Streaming. http://ajaxpatterns.org/Comet.

[11] V. Nazarov. GHCJS Haskell to Javascript Compiler. https://github.com/ghcjs/ghcjs.

[12] M. Snoyman. Client Side Yesod, an FRP-inspired approach. http://www.yesodweb.com/blog/2012/04/client-side.

[13] M. Snoyman. Warp: A Haskell web server. *IEEE Internet Computing*, 15(3):81–85, 2011.

[14] J. Stutterheim. Improving the UHC JavaScript backend. Technical report, Utrecht University, 2012.

[15] A. Vermeulen. On Getting Rid of JavaScript. Technical report, Utrecht University, 2012.