On Getting Rid of JavaScript

Alessandro Vermeulen (3242919)
Department of Computer Science, Utrecht University

February 23, 2012

1 Introduction

Haskell is a beautiful and powerful language. Its functional background and strong type system makes it particularly useful for programming complicated computations and programs. Although Haskell is intrinsically well suited to program calculations it is, from a programmer's standpoint, harder to write a application with a native GUI in it. Currently there are only two options when writing a GUI: either use `wxHaskell` or `GTKHaskell`. Both bindings to the GUI suffer however from being not well maintained and in being difficult to set up.

One solution to work around needing a complicated binding to a GUI library is to create a website. This website will look and work (virtually) the same across multiple operating systems and browsers. It is then the browser that will take care of providing the correct, operating specific, look and feel to the website.

Traditionally websites are 'programmed' in `HTML` and JavaScript.

This project was all about showcasing the UHC's JavaScript-backend. The idea is to get rid of the necessity of having to write your own JavaScript. JavaScript is a language that contains idiosyncrasies, such as the changing of the scope of *this*. On top of that although it is an imperative, object-oriented, programming language with support for first class functions, it does not contain convenient ways of partial application, and lazy evaluation. All essential elements of Haskell. In addition it is not a statically typed language so mistakes are easy to make.

In the following section we will provide some basic knowledge. The reader accustomed with the UHC JavaScript backend may skim ahead to section 3 where we describe the goal of this project. In section 4 we describe the web application itself. In section 5 we describe the additions made to the `uhc-jscript` library. After this some of the more interesting encountered problems are described in section 6. Section 7 will describe future work and section 8 will show how to set up the application. Finally we will conclude with the conclusion in section 9.

2 Prelude

The FFI in UHC supports exporting and importing functions to and from JavaScript. It is named the "jscript calling convention". An import looks like the following:

```
foreign import js "window.location.url"
  windowLocationURL :: IO JSString
```

This makes the property *window* ∘ *location* ∘ *url* available as a Haskell function named *windowLocationURL*. Exporting functions is also possible by using **export** instead of **import**:

```
foreign export jscript "fib" fib :: Int → Int
```

Exporting an Haskell function will wrap it in a native JavaScript function that will force evaluation.

For more information on the UHC FFI and the UHC Foreign Expression Language read "Improving the UHC JavaScript backend"[9].

The most used function in jQuery is the *jQuery* function and its alias `$`. It is available from module `Language.UHC.JScript.JQuery.JQuery` in the `uhc-jscript` [10] as *jQuery* :: *String* → *IO JQuery*. Consider the following HTML snippet, it represents a button.

```
<input type="button" id="foo" value="Click me!"/>
```

Adding an on-click event to this button is quite easy. With the following code you add an on click event that will show an alert box with the text "Hello world!".

```
main = do
  button ← jQuery "input#foo"
  let eventHandler _ _ = alert "Hello world!"
  bind button Click eventHandler
```

In the code example above the button is requested by its CSS selector, processed through jQuery to add the jQuery functionality (by *jQuery*), then we create an event handler that will always show an alert box with the text "Hello world!". Finally we add this event handler to the button as a function to execute when we click on the button. The *bind* function takes care of converting the Haskell function to a JavaScript function before registering as a callback for the JavaScript

event.

3 The goal

This project is a case-study to find out whether programming a website in Haskell using the UHC JavaScript backend is feasible. And to in the process add additional features to the backend or register them for future implementation.

4 The application

In this project I have ported the 'JCU app'[11], a teaching assist tool for Prolog written at Utrecht University. It was written using `Brunch` [2] and `coffeescript` [4]. `Brunch` contains a library to add models and views to JavaScript (called BackBone), and `coffeescript` is a syntactic sugar language for JavaScript.

The application consists of an input box where one can enter the Prolog term to be 'proven' on the left hand side and a list of rules that are available in the environment on the right hand side. The student can prove the statement by dropping a rule from the environment on the tree on the left. Each node (term) will expand according to the definition of the rule if the term in the node and the rule are unifiable. In addition the student can perform manual substitutions by entering a variable and its substitution. During this process the student can hit "Check Proof" to see whether he has made any mistakes.

In according with the goal stated above the idea was to port the 'JCU' app as directly as possible, minimizing the areas where things could go wrong. Rewriting the application to use the full extent of Haskell's language constructs is left for future work. See section 7.4 for a more detailed description.

In the porting process I started with moving the templates such as they were stored in `Brunch`. (They are stored in separate files.) When porting these templates I had to take into account that `Brunch` wraps each view or template inside another element. Currently the templates are written down as a single line with all quotes (") escaped. This way of including templates is inconvenient as it will make it harder to maintain them. See also section 6.4.

Next on the list was retrieving the rules stored on the server through an `AJAX` request. For this the `uhc-jscript` library needed extensions. See section 5 for more on this. In the original application the requests were made blocking which among other things also meant that the function call could *return* its result but it also meant blocking any DOM operations or other JavaScript evaluating. In the port we switched to proper `asynchronous` calls. This meant that functions containing these calls had to be split up in two parts. One part for gathering

and validating the necessary data and the second part for handling the result of the call to the server.

The interesting part happens in the proof tree. In the original application there was a global store containing the tree. From this the tree was drawn in HTML and labels indicating the level and element, in essence the path to the node in the tree, of the node were attached as id to the input elements. When a rule is dropped on one of the terms in the proof tree it would lookup this path, and send it to the server together with the proof tree for checking of the validity of the proof.

Instead of this detour we now directly try to validate the proof in the browser. This is possible because the `uu-tc` and `NanoProlog` library are compilable with UHC to JavaScript. A side-effect of this is that the web application and indeed the whole 'tab' in the browser may hang when the proof checking shoots into infinite recursion. Section 6.2 describes this problem in more detail. The event handler for handling the event when a rule is dropped on the proof tree is partially applied with the current proof tree, as an alternative to looking it up somewhere in globally existing tree.

When a rule is dropped they are unified and if that succeeds the expanding of the rule happens, resulting in a new proof tree that is then drawn again, together with the updated event handlers containing the new proof. An interesting issue regarding the *this* keyword occurred and is described in section 6.1.

The result of the port is a 300 lines long Haskell file containing also comments, as well as several additions to the `uhc-jscript` library, see section 5.

4.1 The advantages

The advantages of programming the client side in Haskell are numerous. One of the biggest advantages is being able to use Haskell code in your web applications with the same semantics as you are used to from your normal Haskell programs. In our case it was almost no work at all to reuse the code from the server side in the client app. This meant it was possible to drop in the `uu-tc`[7] parser and the NanoProlog library. Also section 7.1 describes a possibility we can take advantage of. By using `uu-tc` and the `NanoProlog` library directly in the client application we eliminated the need to communicate with the server for all but storing the rules on the server.

One thing to look out for however is the limited support of post Haskell'98 of UHC. Not all advanced features from GHC are available such as type level computations, e.g. type families, so not every package will work with UHC. More importantly Cabal and UHC don't play nice together at the moment. This means you might have to do some additional work to get a hackage package included in your UHC program by unpacking and telling the compiler where to look manually.

The JCU application used some GHC only features in the same files as where it defined its types for handling Prolog. Thus the types and Prolog proof checking code have been copied from the JCU application instead of importing the module.

Additionally the client-side application now always works as expected. In the original application the drag-and-drop did not always work on the first load.

4.2 Disadvantages

The only disadvantage that can be noted is that the interaction with the user has become a tiny bit slower. It now mainly depends on the computer's speed instead of the server's speed and latency of the connection to the server. This because all computations are now run in JavaScript on the client. The upside is that working on large proof trees does not slow down the interaction much. For the end-user (the student) this might even be an improvement as connections to the server tend to be slow, and because a lot of students use the program simultaneously, e.g. during class. The 'slow' performance in the client-side code could also be alleviated by performing heavier optimizations during compilation from Haskell to JavaScript, by improving the efficiency of the RTS, and by improving the efficiency of the JavaScript engine itself.

5 Additions to the `uhc-jscript` library

As the original application uses `coffeescript` and `Brunch`, as well as jQuery and jQuery UI to program the user interface, we need to create bindings to jQuery and jQuery UI. We decided to bind to jQuery and jQuery UI directly as you would do that when writing plain JavaScript and because writing it in Haskell from scratch would take too much time. Just binding to `Brunch` would be to little work to be interesting. So in order to create the port the existing bindings to JavaScript and especially to jQuery and jQuery UI needed to be improved and extended. Bindings to as well as abstractions over the jQuery's Ajax functionality have been added. This has been done as well for the Ajax Queue [8], jQuery UI Draggable, jQuery UI and Droppable libraries have been added.

When communicating through AJAX/JSON you have to pass along JavaScript values. However when you want to use you have to parse them into Haskell values. (In order not to change to much on the server-side and to prevent it from being to dependent on the Haskell runtime representation we decided to keep communicating in JSON.)

To ease converting Haskell values from and to JavaScript values there exist two classes: *FromJS* and *ToJS*. One can easily see that these functions (*toJS* and *fromJS*) are not safe as they do not have a way to express failure.

```
class ToJS a b where
  toJS :: a → b
class FromJS a b where
  fromJS :: a → b
```

A 'safe' *FromJS* class has also been added called *FromJSPlus*. This works by checking the JS type of the object you are about to convert. This behaviour is the default implementation and can be replaced by another. It resembles the *Typeable* class but uses a simpler type representation that directly maps to JavaScript's type system.

```
class FromJS a b ⇒ FromJSPlus a b where
  jsType :: a → b → String
  check :: a → b → Bool
  check a b = jsType a b ≡ fromJS (typeof a)
  fromJSP :: a → Maybe b
  fromJSP a = let (v :: b) = fromJS a
    in if check a v then
         Just v
       else
         Nothing
```

The $(v :: b)$ is necessary to restrict the type of v to a monomorphic type, UHC is not able to discover this on his own accord.

An example here would be the conversion from a JavaScript string to Haskell string.

```
instance FromJSPlus JSString String where
  jsType _ _ = "string"
```

When you use *fromJSP* instead of *fromJS* you are able to catch the error when something went wrong, the default implementation of check just checks the JavaScript type but you could also change the functions such that it would perform a real 'parse' of the value.

6 Encountered issues

During the port of the program I encountered the following issues, I will describe their origin and we will see the solution offered.

6.1 The annoying wandering scoped *this*

A common and well-known problem and powerful semantic when programming in JavaScript is that the scope of *this* is relative to where the function containing the reference to *this* is being executed. This can and indeed does lead to many cases of unexpected behaviour.

The jQuery UI library ensures that *this* in the event callbacks points to the object you, e.g., dropped another object on, instead of passing it as an argument. However, in our RTS we wrap functions in other objects, thus changing the scope of *this*. In order to work around this I've added the following JavaScript code to the system. As this function will be called by the event system the *this* in the code refers to the event object. By pushing it onto the front of the arguments array we copy it and thus make it available for use further on in our Haskell code.

```
function wrappedThis(cps) {
  return function() {
    var args = Array.prototype.slice.call(arguments);
    args.unshift(this);
    return cps.apply(this, args);
  }
}
```

The code is agnostic of the amount of parameters the *cps* parameter needs but just pushes the *this* to the front of the argument list. This is reflected in the wrapper on the Haskell side as follows:

```
type UIEventHandler       =                JQuery → JUI → IO Bool
type UIThisEventHandler   = JQuery → JQuery → JUI → IO Bool
type JUIThisEventHandler = JSFunPtr UIThisEventHandler
foreign import js "wrappedThis(%1)"
  wrappedJQueryUIEvent :: JUIThisEventHandler
                       → IO JUIEventHandler
```

The function *wrappedJQueryUIEvent* will call wrappedThis, that in essence partially applies the provided function with *this*. And thus its result type is *JUIEventHandler*.

Instead of wrapping functions when you need them to access *this* manually by calling *wrappedThis* the wrapping of *this* can also already be done when dynamically creating the JavaScript function. That is directly in the the function that is generated by calling the imported 'wrapper'. The wrapper would then always output a function that receives an extra *this* argument. The disadvantage here

is that it will introduce extra overhead, however it could potentially cause less unexpected behaviours. Additionally the type of the *this* argument is not always known so it might not be the correct place to do this. One step further would be to change the *JSFunPtr* type to always include a *this* argument.

The following code shows how to add make list items droppable. That means, make them able to receive an item.

```
...
dropzones ← jQuery "li"
drop      ← mkJUIThisEventHandler (λthis event ui → return True)
drop'     ← wrappedJQueryUIEvent drop
droppable dropzones $ Droppable (toJS "someAllowedSelectorClass")
                                drop'
...
```

Providing the *this* element as a parameter would give less cause for the user of the `uhc-jscript` library to import the keyword *this* into their application and use it in the same way as they would have used in the JavaScript counterpart of the function as shown below. (The *return True* is the *noop* operation for jQuery events.)

```
foreign import js "this"
  getThis :: IO (JSPtr a)
eventHandler = do
  this ← getThis
  doSomething this'
```

In the example above *this* will point to an object of the Haskell RTS and not, e.g., the object that received a draggable object.

6.2 (Possibly) Non-terminating code

As our program includes a checker for Prolog proofs our application contains code that might not terminate as is in general not possible to check a Prolog proof without shooting into infinite recursion. (Try to proof the term f with the rule $f \vdash f$, reads as you can prove f if you have f.)

In the original setup all the (heavy) calculation was done by the server. The server also is able to set a timeout on calculations, or rather, requests in general. As JavaScript is single threaded (mainly) the infinite checking of the proof would block the whole server. In the hope that with *Deferred* from jQuery would solve this I added the following JavaScript code to the lib and created a binding to it in `Language.UHC.JScript.JQuery.Deferred`.

```
/**
 * boundExecution :: JSFunPtr (IO a) -> JSFunPtr (IO a)
                   -> Int
                   -> JSFunPtr (a -> IO b)
                   -> JSFunPtr (a -> IO b) -> IO ()
 */
function boundExecution(calc, fallback, timeout, onC, onF) {
  var dfd = new jQuery.Deferred();

  var tryBranch = function() {
    var res = calc();
    dfd.resolve(res);
  };

  var fallBackBranch = function() {
    var res = fallback();
    dfd.reject(res);
  }

  setTimeout(fallBackBranch, 5 + timeout);
  setTimeout(tryBranch,      5);


  $.when( dfd.promise() ).then(onC, onF);
}
```

This still doesn't work in our case as the checking doesn't pause at any time or use *setTimeout*. Suppose that the code was written in a CPS[3] monad one could use that to introduce *setTimeout*s in the JavaScript code with a minimal interval. This would most likely create a situation that would work with *jQuery* ∘ *Deferred* when used in the same fashion as the above code example.

WebWorkers A better, more suited, approach is to use Web Workers[12]. WebWorkers are a part of the new HTML5 standard. These web workers work in a background thread. Using them should solve our problem as they will work in the background and can be killed. This way we can put a timeout on the checking. I've tried to walk this path and added the `Language.UHC.JScript.JQuery.WebWorker` file to the `uhc-jscript` library for it. However, the crucial point here is the passing of data. The messages that pass along the channel between the Worker and the main thread should not contain functions. As our thunks contain functions they cannot be send directly.

Attempts to serialize the Haskell thunk layout using *JSON.stringify* failed as it

does not support serializing functions. In essence the thunks need either to be fully evaluated or the serialization should be written in Haskell and is left for future work.

Working around it Currently proof checking is disabled by default when the user opens the application. The button "Check Proof" will toggle the checking of the proof. This circumvents the checking of intermediate non-checkable proofs.

6.3 Global state

Instead of using a *State* or *Reader* monad containing an *IORef* to save whether proof checking should be done or not I use an hidden input field as a store. In the future this could perhaps be changed to run everything in the *State* or *Reader* monad. However, depending on the application this might add a lot of extra *liftIO* calls to lift our normal calculations back into *IO*.

6.4 Miscellaneous

Among the small other issues I encountered are the following:

Overlapping instances are a bit tricky in UHC. If you happen to create such a condition you will get the error that an instance does not exist at all! So watch out for this!

Missing functional dependencies cause you to have to write down additional type signatures. Such as in the following scenario. Without the *::String* annotation the compiler will give you an error.

```
do jsRuleText ← smth :: IO JSString
   let ruleText = fromJS jsRuleText :: String
   ...
   case tryParseRule ruleText of
```

Including large strings of text Currently it is not possible (per the Haskell standard) to include multi-line strings or to add strings containing " without escaping them.

```
jcu.hs:139-152:
  Predicates leading to ambiguous type:
    preds                : Language.UHC.JScript.Types.FromJS
```

```
                              UHC.Base.PackedString v_30_3523_2:
bindings                : $ok: MONO  [v_30_3559_0_0] ->
                                 MONO  UHC.Base.IO MONO  UHC.Base.Bool
bindings (quantified): $ok: MONO  [v_30_3559_0_0] ->
                                 MONO  UHC.Base.IO MONO  UHC.Base.Bool
```

Literals in FFE are possible when using ‘” see the following example.

```
foreign import js "$('document').ready(%1)"
  _ready :: EventHandler → IO ()
```

Creating new objects from Haskell is a bit of an hassle. The literal trick from above doesn't work as you will end up with “new Worker” between string quotes in JavaScript.

```
foreign import js "'new Worker'(%1)"
  _newWorker :: JSString → IO WebWorker
```

A solution would be to add the *new* keyword and syntax to the `FFE`.

7 Future Work

7.1 Communicating with the server

Currently the communication with the server is encoded manually. That is, the creation of the JavaScript values to be send over hardcoded in the application. When coding both the server and the client in Haskell one should be able to create something like ‘typed channels’ for each server endpoint. This would further improve the type safety of the application.

7.2 Background threading

Although the basic interface for using `WebWorkers` is present it is currently not possible to seamlessly pass Haskell values due to the presence of functions. Figuring out how then to pass Haskell values is subject to future research.

7.3 Generic *FromJS* and *ToJS* for converting objects/records

It should be fairly straightforward to implement a generic implementation for *FromJS* and *ToJS* using deriving Generic.

7.4 Providing an api to build web applications

One could think of providing a similar API such as WxHaskell[5] does for constructing native application, but for web applications. Also one could think of providing a Functional Reactive[6][13][1] interface to building the web application.

8 Locations and setup

The UHC can be found here: `http://www.cs.uu.nl/wiki/bin/view/UHC/`. The JCU application can be found here: `https://github.com/spockz/JCU`. The client-side project is located in `/resources/static/hjs`. The `uhc-jscript` can be found here: `https://github.com/spockz/uhc-jscript`.

The makefile in `hjs` assumes the presence of a `$UHC` variable containing the location of the UHC. Additionally it expects the following three variables: `UHC_JSCRIPT`, `UHC_NANOPROLOG`, and `UHC_UU_TC` to point to the source directories of respectively: uhc-jscript, NanoProlog (`https://github.com/spockz/NanoProlog`), and the Utrecht Talen & Compilers Parser Library (`http://www.cs.uu.nl/wiki/bin/view/TC/CourseMaterials#Parser_Combinator_Library`). A package for is also available from hackage, maintained by Jurriën Stutterheim.

Be sure to have a PostgreSQL server running. You can configure the connection to the server in the file `JCU/config/connection_string.conf`. The script to create the tables is located in `JCU/packaging/initPgSqlDb.sql`.

9 Conclusion

We have seen that it is possible to program the client-side of a web application in Haskell and that it is not a lot of work. Partial application can be used as an alternative for global variables. Not only can you program simple web applications but you can also include parser combinator libraries into them. On top of it all the performance is relatively good. There is still room for progress regarding improving the performance. It is possible to not have to write your own JavaScript code.

References

[1] Heinrich Apfelmus. *Reactive Banana*. URL: {http://www.haskell.org/haskellwiki/Reactive-banana}.

[2] *Brunch.IO*. URL: {http://brunch.io/}.

[3] Koen Claessen. "A poor man's concurrency monad". In: *J. Funct. Program.* 9 (3 May 1999), pp. 313–323. ISSN: 0956-7968. DOI: http://dx.doi.org/10.1017/S0956796899003342. URL: http://dx.doi.org/10.1017/S0956796899003342.

[4] *CoffeeScript*. URL: {http://coffeescript.org/}.

[5] et al. Daan Leijen. *WxHaskell*. URL: {http://www.haskell.org/haskellwiki/WxHaskell}.

[6] Conal M. Elliott. "Push-pull functional reactive programming". In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. Haskell '09. Edinburgh, Scotland: ACM, 2009, pp. 25–36. ISBN: 978-1-60558-508-6. DOI: http://doi.acm.org/10.1145/1596638.1596643. URL: http://doi.acm.org/10.1145/1596638.1596643.

[7] Andres Löh, Johan Jeuring, and Doaitse Swierstra. *UU-TC: Haskell 98 parser combintors for INFOB3TC at Utrecht University*. URL: {http://www.cs.uu.nl/wiki/bin/view/TC/CourseMaterials}.

[8] Oleg Podolsky. *jquery-ajaxq*. URL: {http://code.google.com/p/jquery-ajaxq/}.

[9] Jurriën Stutterheim. *Improving the UHC JavaScript backend*. Tech. rep. Department of Information and Computing Sciences, Utrecht University, 2011.

[10] Jurriën Stutterheim, Alessandro Vermeulen, and Atze Dijkstra. *UHC-JScript library*. URL: {https://github.com/spockz/uhc-jscript}.

[11] Wouter Swierstra, Doaitse Swierstra, and Jurriën Stutterheim. *Logisch en Functioneel Programmeren voor Wiskunde D*. Tech. rep. UU-CS-2011-033. Department of Information and Computing Sciences, Utrecht University, 2011.

[12] W3C. *Web Workers*. URL: {http://dev.w3.org/html5/workers/}.

[13] Zhanyong Wan and Paul Hudak. "Functional reactive programming from first principles". In: *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 242–252. ISBN: 1-58113-199-2. DOI: http://doi.acm.org/10.1145/349299.349331. URL: http://doi.acm.org/10.1145/349299.349331.