Building JavaScript Applications with Haskell

Atze Dijkstra, Jurriën Stutterheim, Alessandro Vermeulen, and Doaitse Swierstra

Department of Information and Computing Sciences
22 Universiteit Utrecht
23 P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{atze, j.stutterheim, a.vermeulen, doaitse}@uu.nl

Abstract. We introduce the Utrecht Haskell Compiler (UHC) JavaScript backend; a compiler backend which allows one to cross-compile Haskell to JavaScript, so it can be run in the browser. To interface with JavaScript and overcome part of the impedance mismatch between the two languages, we introduce the Foreign Expression Language; a small subset of JavaScript for use in Foreign Function Interface (FFI) imports. Finally we discuss the implementation of a JavaScript application, completely in Haskell, with which we show that it is now possible to develop JavaScript applications completely in Haskell.

1 Introduction

When developing interactive web applications, JavaScript is often the language of choice, due to every major browser supporting it natively. In contrast to other client-side programming languages, no plugins are needed to execute JavaScript. Unfortunately, JavaScript is currently the *only* client-side programming language that is supported on all major browsers. People wishing to use other programming languages or paradigms have to rely on using existing plugins such as Flash or Java Applets, writing custom browser plugins, or hacking the browsers themselves. None of these options are ideal, since they either require a lot of work, or force the use of strict, imperative languages. Instead of choosing between the aforementioned options, we use the Utrecht Haskell Compiler (UHC) [10, 11] to compile Haskell code to JavaScript, effectively turning JavaScript into a high-level byte-code of some sorts.

In this paper, we introduce the UHC JavaScript backend, a compiler backend that allows one to compile Haskell to JavaScript, while keeping Haskell's lazy semantics. To overcome the impedance-mismatch between Haskell and JavaScript, we have extended UHC's Foreign Function Interface (FFI) with a small JavaScript-like expression language we call the Foreign Expression Language (FEL). With these enhancements to the FFI, we claim that it is now possible to write complete JavaScript applications using only Haskell. We back this claim up by porting a web-based Prolog proof assistant from JavaScript to Haskell. While this paper

focusses on Haskell, the ideas should be relatively easy to implement in similar languages, such as Clean.

With this paper, we make the following contributions:

- We introduce the UHC JavaScript backend; a compiler backend that allows one to compile any Haskell code supported by the UHC to JavaScript and execute it in the browser with the same semantics.
- We introduce the Foreign Expression Language (FEL), which allows for a more natural way of interfacing with object-oriented languages.
- We show that it is now possible to write complete JavaScript applications using only Haskell.

The rest of this paper is structured as follows: section 2 introduces the UHC JavaScript runtime system (RTS) and FFI, after which section 3 shows the implementation of a complete JavaScript application in Haskell, after which sections 4 and 4.4 discuss future and related work respectively. Finally, section 5 concludes.

2 Compiling Haskell to JavasScript

2.1 Runtime System

There exists an obvious mismatch between Haskell and Object-Oriented (OO) languages, which has been addressed in various ways over time (Section **??**):

- Mapping the runtime machinery required for Haskell to an imperative language has to deal with the lazy evaluation strategy imposed by Haskell (rest of this section).
- Use of OO language mechanisms as available in JavaScript, in particular prototype based objects; we avoid dealing with this problem.
- Use of available JavaScript libraries; we deal with this in the next section by exploiting the freedom offered by Haskells Foreign Function Interface (FFI)

The design of any backend for a lazy functional languages needs to deal with functions, their (lazy) application to arguments, and evaluating such applications to Weak Head Normal Form (WHNF). The design should also cater for under- and oversaturated function applications as well as tail recursion.

In UHC's JavaScript backend functions and their applications are both represented straightforwardly by objects:

$$
\begin{array}{l}
\mathit{Fun.prototype} = \{ \\
\quad \mathit{applyN} : \textbf{function}\ (\mathit{args}) \ldots \\
\quad \mathit{needsNrArgs} : \textbf{function}\ () \ldots \\
\}
\end{array}
$$

```
function Fun (fun) {...}
```

For now we omit implementation details and only expose the programmatic interface as used by the runtime system. A *Fun* object wraps a JavaScript function so that it can be used as a Haskell function. The *applyN* field is only used when function applications are being evaluated (forced); only then it is necessary to know the *needsNrArgs* number of arguments which must be passed. For the time being it stays unevaluated as a *Fun* object wrapped inside an *App* or *AppLT* closure object.

Similarly, partially applied (and thus undersaturated) functions need to store already passed arguments and how many arguments are still missing. An *AppLT* (*LT* stand for *less than*) object encodes this and again we provide its programmatic interface first:

```
AppLT.prototype = {
  applyN : function (args) ...
  needsNrArgs : function () ...
}
function AppLT (fun, args) {...}
```

An *AppLT* only wraps other *AppLT* objects or *Fun* objects.

Finally, for all remaining saturation cases an *App* object is used, knowledge about saturedness is delegated to the encapsulated function object, which may be another *App*, *AppLT*, or *Fun*.

```
App.prototype = {
  applyN : function (args) ...
}
function App (fun, args) {...}
```

With this interface we now can embed Haskell functions; for example the function $\lambda x \rightarrow id(id\ x)$ is, assuming an elementary JavaScript function *id* is available, by:

```
new Fun (function (x) {
  return new App (id, [new App (id, [x])]);
})
```

Evaluation is forced by a separate function *eval* which assumes the presence of an *eOrV* (evaluator Or Value) field in all Haskell runtime values, which tells us whether the JavaScript object represents a Haskell non-WHNF value which needs further evaluation or not; in the former case it will be a JavaScipt function of arity 0, which can be called. A Haskell function or application object does not evaluate itself since the entailed tail recursion will cause the stack of the underlying JavaScript engine to flow over. The separate external function *eval* doing the evaluation allows non WHNF values to be returned, thus implementing a trampoline mechanism:

```
function eval (x) {
  while (x ∧ x.eOrV) {
    if (typeof x.eOrV == 'function') {
      x = x.eOrV ();
    } else {
      x = x.eOrV;
  } }
  return x;
}
```

Even normal JavaScript values can be thrown at *eval*, provided they do not (accidentally) contain an *eOrV* field. The actual *eval* function is somewhat more involved as it provides some protection against null values and also updates the *eOrV* field for all intermediate non WHNF objects computed in the evaluation loop.

As usual the evaluation is driven by the need to pattern-match on a value, e.g. as the result of a case expression or by a built-in JavaScipt primitive which is strict in the corresponding argument such as in the wrapper of the primitive multiplication function, which contains the actual multiplication (∗):

```
new Fun (function (a, b) {
  return eval (a) ∗ eval (b);
})
```

Depending on the number of arguments provided either an undersatured closure is built, or the function is directly invoked using JavaScripts *apply*. In case too many arguments are provided a JavaScript closure is constructed, which subsequently is evaluated in the evaluation loop of *eval*. The implementation of *AppLT* is similar to that of *Fun*. *App*s implementation of *applyN* simply delegates to *applyN* of the function it applies to. Also omitted are the encodings of nullary applications, used for unevaluated constants (CAF, Constant Applicative Form) and indirection nodes required for mutual recursive definitions. Data types and tuples are straightforwardly mapped onto JavaScript objects with fields for the constructor tag and its fields. If available, record field names of the corresponding Haskell data type are used.

2.2 JavaScript Foreign Function Interface

Talk about all the cool stuff - Mostly discuss the Foreign Expression Language (FEL)

- Briefly mention the dynamic/wrapper things

3 The JCU Application

What does the app do? How is it implemented? Does it work? Does it work well? Is the UHC JS backend a full replacement for JS coding? Etc? [**?**,**?**] [**?**]

```
exp  ::= '{}'                -- Haskell constructor to JS object
       | (arg | i) post*     -- JS expression
post ::= '.' i               -- object field
       | '[' exp ']'         -- array indexing
       | '(' args ')'        -- function call
args ::= ε | arg (, arg)*    -- possible arguments
arg  ::= '%' ('*' | int)     -- all arguments, or a specific one
       | '"' str '"'         -- literal text
i    ::= a valid JavaScript identifier
int  ::= any integer
str  ::= any string
```

Fig. 1. Import entity notation for the JS calling convention

The JCU Prolog Proof Assistant[**?**] is a tool developed for students to familiarize themselves with Prolog through the aided solving of Prolog proof trees. Originally programmed in `coffeescript` [8] (sugar for JavaScript) and using the `Brunch` [19] framework.

Students can proof their Prolog query by applying a combination of variable substitution and dropping rules from the list of rules on the tree.

The tool is presented as a web application and runs in any modern browser. The application is written in Haskell which compiles to JavaScript by UHC. It uses the *uhc-jscript* [28] library. The *uhc-jscript* library provides (among others) bindings to the jQuery[24] JavaScript library and the AjaxQueue[23] library.

As our original application was written

3.1 Implementation Issues

Which issues do you run into in general when you develop a JS app in Haskell?

Ale: non-termination/threads, no native OO, interfacing with annoyingly multi-interpretable functions (functions that are dimensioned in both the number and type of their arguments)

3.2 Performance

Performance is decent, except for the actual Prolog backtracking. Where is the bottleneck? Ale: Probably as I've stated before the problem is in the memory management as is supported by the benchmarks in Chrome and Fx.

4 Future Work

4.1 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.

4.2 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.

4.3 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.

4.4 Providing an api to build web applications

One could think of providing a similar API such as WxHaskell[**?**] does for constructing native application, but for web applications. Also one could think of providing a Functional Reactive[13, 31, **?**] interface to building the web application.

Typechecking Currently, foreign expressions are not typechecked at all. In case of a programming error the compiler will currently panic in the best-case scenario and happily generate JavaScript code that fails at runtime in the worst-case scenario. Constraints on the imports and exports need to be formalised and the foreign types should be typechecked according to the foreign expressions. Some example constraints that could be typechecked:

- Only datatype values may be exported as objects, not functions or primitive types.
- Only wrapped functions may be exported in objects.

The first item could be realised by supporting type constraints in the foreign import. This is already allowed by the type system, but the RTS does not currently support this.

Testing During development, the code has only received limited, informal testing. The UHC's test suite should be expanded to enable automated testing. Additionally, real-world JavaScript front-end applications should be ported to Haskell and the *uhc-jscript* library to identify shortcomings of the existing ideas and implementations.

Benchmarking an optimising Some benchmarks have been performed that show that the generated JavaScript is quite a bit slower than a hand-written version of the same code. Unfortunately, these numbers aren't publicly available, nor is is clear where the biggest bottlenecks are located. It would be very interesting to do another round of benchmarks, including the object code. Afterwards, it would be interesting to identify bottlenecks and find ways to speed up the generated code.

Static object compilation Currently the only way of converting a datatype to a JavaScript object is to do so at runtime. This, however, is a process with time complexity linear in the number of datatype records. Future work could focus on generating (parts of) JavaScript objects at compile-time, so that only dynamic values will need to be copied to the object at runtime.

Numeric object indices When a datatype without record selectors is converted into a JavaScript object, the object's attribute name becomes an integer $\geqslant 1$ with an underscore prefix. Ideally this would be a numeric index $\geqslant 0$.

Other approaches. The idea of running Haskell in a browser is not new. To our knowledge first attempts to do so using JavaScript were done in the context of the York Haskell Compiler (YHC) [3]. The Document Object Model (DOM) inside a browser was accessed via wrapper code generated from HTML standard definitions [2]. However, YHC is no longer maintained and direct interfacing to DOM nowadays is replaced by libraries built on top of the multiple DOM variations.

The idea of running functional programs in a browser even goes further back to the availability of Java applets. The workflow framework *iTasks*, built on top of the Clean system [5], uses a minimal platform independent functional language, SAPL, which is interpreted in the browser by code written in Java. The latest interpreter incarnations are written in JavaScript [14, 9, 22]. Although currently a Haskell frontend exists for Clean, the use of it in a browser seems to be tied up to the iTasks system. The intermediate language SAPL also does not provide the facilities as provided by our Haskell FFI.

Of the GHC a version exists which generates JavaScript [21], based on the GHC API, supporting the use of primitives but not the FFI. Further down we elaborate on some consequences of multiple platforms and backends relevant for this GHC backend variant as well.

Both "Functional javascript" [26] and "Haskell in Javascript" [4] do not use a separate Haskell compiler. Instead, JavaScript is used directly in a functional style, respectively a small compiler for a subset of Haskell has been written in JavaScript.

Object orientation. Object Oriented behavior itself can be realized inside Haskell by exploiting the class system [25, 15]. However, we aim to access libraries written in JavaScript, not mimic JavaScript or OO mechanism in general inside Haskell.

However, when functionality of the libraries would have to be (re)written in Haskell some form of OO mechanism should be available. This issue arises when one would code in Haskell a platform independent part of an otherwise OO GUI library, say *wxHaskell*. For now we limit ourselves to accessing JavaScript libraries via the FFI, hiding OO idiom inside FFI import entities.

Type system absence. JavaScript has no type system, the absence of which can be dealt with by using phantom types in wrapper functions around untyped FFI calls. More problematic are for example DOM functions returning objects with a different interface, like a DOM element or attribute. A sum type defining all possible result types could be used, but data types are not extensible, which might be too limiting in practice. Dynamics might be used as result values, but require assistence from the runtime system as well as knowledge about types (e.g. in the form of *TypeRep*). Existentially quantified data types and classes might be used (similar to extensible exceptions [16]), but then knowledge about the class system also seeps into the runtime system. Currently this has not yet been further addressed.

Side effects. All access to JavaScript values is done in the IO monad, so side effect can be properly modelled. For now it is assumed that no threads exist. Since JavaScript's worker thread mechanism can be used safely we currently do not need semaphores, STM, or other shared data access machinery. Some values like the globally available `window` in a browser could be accessed without the use of the IO monad because its value does not change. However, if and when this assumption would not hold in a near future it would break our wrapping code as well.

Deployment. JavaScript code is usually downloaded, hence compact representation as well as avoiding or delaying loading of code is important. UHC allows pruning of unused code as to achieve a relative compact representation, but provides no mechanism for dynamic loading. It is left up to the user of the compiled code to incorporate it into a webpage.

Mapping to OO runtime environments. In general, it is attractive to map onto available virtual machines for popular OO languages, in particular because of the availability of a wealth of libraries. Targeting the Java virtual machine (JVM)

has been experimented with [30, 12, 27, 29], as well as with the `.NET` platform [20, 1, 17, 18]; UHC also provides a backend to the JVM [12] using the same technique as described here. However, interfacing to libraries is still lacking in UHC, and in general library access and other runtime oriented features (threads, exceptions) is where the real work lies [6]. Wrapping and interfacing to libraries has to be done manually or by tools interpreting library code which requires substabtial effort and is suffering from misinterpretation. In the case of JavaScript lack of typing annotations even precludes automatic FFI wrapper generation, unless type annotations in comments could be trustworthy and formal enough to be used instead.

Efficient code generation is also an issue. Usually non standard OO language constructs are used to implement standard idiom of (lazy) functions. For now, with UHC we have taken the approach to first make it work and not bother about efficiency, generating code from an early stage in the compiler pipeline. We expect exploitation of the results of a strictness analyser to speed up the code considerably, especially because the existing JavaScript compilers to be better able to analyse the provided code.

Libraries, Haskell platform. Targeting Haskell to a different platform means that some assumptions following from using a single platform only are no longer valid. First, a different platform means a different runtime environment. Almost all of the UNIX functionality is available for the usual Haskell UNIX runtime, but is naturally not available inside a web browser and, vice verse, specific JavaScript libraries like jQuery are not available on a UNIX platform. Some library modules of a package (partially) cannot be build on some platforms, while others (partially) can. To cater for this, UHC rather ad-hoc marks modules to be unavailable for a backend by a pragma `{-# EXCLUDE_IF_TARGET js #-}`. Of course *cpp* can still be used to select functionality inside a module. However, in general, awareness of platform permeates all aspects of a language system, from the compiler itself to the library build system like *Cabal*. In particular, *Cabal* needs a specification mechanism for such variation in target and platform to allow for selective compilation of a collection of variants. Currently this means that UHC compilation for the JavaScript backend cannot be done through *Cabal*.

A second aspect has more to do with the evolution of Haskell as an ecosystem. Many libraries go far beyond the Haskell standard by making use of a plethora of GHC extensions. Currently, such libraries evolve to use (say) type families, a feature not (yet) available in UHC. For (non GHC) Haskell compiler writers to keep with this pace of evolution poses a considerable challenge; yet in our opinion there is value in the availability of compiler alternatives as well as variation in what those compilers are good at.

More info, download. For the variant of the JCU application as implemented for this paper more info (download, how to install, etc) is available, via the UHC www site [11] or directly [7].

5 Conclusion

We have shown that the UHC is capable of supporting the development of complete client-side web applications. This opens the door to Haskell-only web development. Better abstractions are still required to reduce the amount of code that lives in the *IO* monad directly, and to give programming with the UHC JavaScript backend a more functional feel. While in most cases performance is acceptable, it needs to be improved if computationally heavy functions are to be run on the client. In order for most of the frequently used Hackage libraries to be run on the client, UHC will need some more work as well.

References

1. The Haskell.net Project. `http://www.cin.ufpe.br/~haskell/haskelldotnet/`, 2004.
2. Haskell in web browser. `http://www.haskell.org/haskellwiki/Haskell_in_web_browser`, 2007.
3. Yhc/Javascript. `http://www.haskell.org/haskellwiki/Yhc/Javascript`, 2007.
4. A haskell interpreter in javascript. `https://github.com/johang88/haskellinjavascript`, 2010.
5. Clean. `http://wiki.clean.cs.ru.nl/Clean`, 2011.
6. GHC FAQ. `http://www.haskell.org/haskellwiki/GHC:FAQ`, 2012.
7. The Utrecht Haskell Compiler JavaScript Backend Page. `http://uu-computerscience.github.com/uhc-js/`, 2012.
8. Jeremy Ashkenas. Coffeescript.
9. Eddy Bruël and Jan Martin Jansen. Implementing a non-strict purely Functional Language in JavaScript. In *Implementation of Functional Languages*, 2010.
10. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The Architecture of the Utrecht Haskell Compiler. In *Haskell Symposium*, 2009.
11. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. UHC Utrecht Haskell Compiler. `http://www.cs.uu.nl/wiki/UHC`, 2009.
12. Atze Dijkstra and S. Doaitse Swierstra. Lazy Functional Parser Combinators in Java. In *Proceedings of 1st Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, pages 11–42. John von Neumann Institute for Computing, 2001.
13. Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.
14. Jan Martin Jansen. *Functional Web Applications, Implementation and Use of Client-Side Interpreters*. PhD thesis, Radboud University Nijmegen, 2010.
15. Oleg Kiselyov and Ralf Lämmel. Haskells overlooked object system, 2005.
16. Simon Marlow. In *Haskell Symposium*, 2006.
17. Erik Meijer and Koen Claessen. The Design and Implementation of Mondrian. In *Haskell Workshop*, 1997.
18. Erik Meijer, Nigel Perry, and Arjan van Yzendoorn. Scripting .NET Using Mondrian. In *ECOOP*, 2001.
19. Paul Miller, Nik Graf, Thomas Schranz, and Andreas Gerstmayr. Brunch.io.

20. Monique Monteiro, Mauro Araújo, Rafael Borges, and André Santos. Compiling Non-strict Functional Languages for the .NET Platform. *Journal of Universal Computer Science*, 11(7):1255–1274, 2005.
21. Victor Nazarov. ghcjs: Haskell to Javascript compiler (via GHC). `https://github.com/sviperll/ghcjs`, 2011.
22. Rinus Plasmeijer, Jan Martin Jansen, and Pieter Koopman. Declarative Ajax and Client Side Evaluation of Workflows using iTasks. In *Principles and Practice of Declarative Programming*, 2008.
23. Oleg Podolsky. jquery-ajaxq.
24. John Resig. jquery.
25. Mark Shields and Simon Peyton Jones. Object-Oriented Style Overloading for Haskell. In *Workshop on Multi-Language Infrastructure and Interoperability*, 2001.
26. Oliver Steele. Functional Javascript. `http://osteele.com/sources/javascript/functional/`, 2007.
27. Don Stewart. Multi-paradigm Just-In-Time Compilation. Master's thesis, The University of New South Wales, School of Computer Science and Engineering, 2002.
28. Jurrin Stutterheim, Alessandro Vermeulen, and Atze Dijkstra. Uhc-jscript library.
29. Mark Tullsen. Compiling Haskell to Java. Technical report, Yale University, Department of Computer Science, 1996.
30. David Wakeling. Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine. In *Programming Languages, Implementations, Logics and Programs*, 1998.
31. 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, pages 242–252, New York, NY, USA, 2000. ACM.