Demo Proposal: GHCJS, Concurrent Haskell in the Browser

Luite Stegeman

stegeman@gmail.com

Abstract

This is a demonstration proposal for GHCJS, a Haskell to JavaScript compiler that uses the GHC API. GHCJS implements many modern Haskell features, including lightweight threading, STM and asynchronous exceptions. The GHCJS runtime has specific provisions to deal with multithreaded Haskell code in the singlethreaded JavaScript runtime.

Keywords Haskell, GHC, GHCJS, JavaScript, web, browser

1. Introduction

The JavaScript problem [1] is well-known in the Haskell world: JavaScript lacks the safety guarantees offered by the Haskell type system, but unfortunately we are stuck with it, since it's the only language that all web browsers support.

Compiling Haskell to JavaScript is an attractive proposition, not only can we rely on Haskell's type system on the client, it also makes it possible to share code between the client and the server.

GHCJS has existed since 2010, created by Victor Nazarov, but it's far from the only solution. A well-known compiler is UHC-JS [2], based on UHC. Alternatives based on GHC are Fay [3] (type checking only) and Haste [4].

GHCJS is the only Haskell to JavaScript compiler with a runtime that fully supports lightweight threads, including black holes, asynchronous exceptions and synchronization through MVar and STM. Additionally, GHCJS supports unboxed arrays, weak references, StableName and a limited form of pointer emulation. A Cabal patch to add GHCJS as a compiler flavor is in the works.

1.1 Haskell to JavaScript

GHCJS uses the GHC API to translate Haskell to JavaScript. It uses STG as its source language, which is translated to JMacro [5]. The translated code is tail call optimized (through a trampoline, since JavaScript does not yet support tail calls natively) and uses a dynamically growing JavaScript array as stack.

Inspired by UHC-JS and Fay, GHCJS supports a convenient *foreign import javascript* syntax:

```
1 foreign import javascript unsafe
2   "$1 + $2"
3   plus :: Int -> Int -> Int
```


Haskell Symposium 2013 23-24 September 2013, Boston

Placeholders *$1* and *$2* are replaced by the actual parameters. Thanks to JMacro, it is possible to use complex JavaScript statements, including local variable declarations and loops, in the import declarations. The local variables are converted to hygienic names, and the foreign import is spliced into the rest of the JMacro AST.

The combined compiled code is then run through a dataflow analyzer to reduce code size by eliminating dead code and redundant assignments. Finally, the GHCJS linker collects all functions reachable by the program and bundles them in a single JavaScript file. Non-Haskell code, for example the RTS and third party libraries, is collected in another file.

1.2 Memory management

Older versions of GHCJS, and also Fay and Haste, use JavaScript closures to represent Haskell heap objects. These have the advantage of being easy to implement. Unfortunately, closures are *opaque*, it is impossible to see from the outside which variables they contain. This makes it very hard to support some Haskell features, for example weak references with finalizers, CAF deallocation and deadlock detection.

Therefore the current GHCJS version does away with JavaScript closures altogether, representing every Haskell heap object [1] as a plain JavaScript object. For example:

```
1  var listElement = { f: Data_List_cons
2                    , d1: value
3                    , d2: nextElement
4                    };
5
6  var someThunk = { f: entryFunction
7                  , d1: someData
8                  };
9
10 var someFunction = { f: theFunction
11                    , d1: freeVariable1
12                    , d2: freeVariable2
13                    };
```

Every heap object has an entry function *f*. For thunks, *f* returns the object reduced to WHNF, for functions, *f* is the function itself. All data specific to a single heap object is contained in the data fields *d1, d2, ...* Analogous to the info table in GHC, *f* contains some metadata about the object, stored in properties of the function object, for example the constructor tag or the arity of the function.

When a shared thunk is entered, *f* immediately overwrites its data by a black hole (eager blackholing) and pushes an update frame on the stack. Since we don't have control over the JavaScript garbage collector, this is the best way to prevent references from lingering around, leaking memory.

Using JavaScript objects as Haskell heap objects means that the JavaScript garbage collector can do most of the memory manage-

[1] There are some exceptions: Types like *Int*, *Char* and *Double* are represented as primitive JS numbers, the first two constructors in enumeration ADTs are represented as the boolean values *false* and *true*

ment work for us. Some Haskell features however, like weak references, require us to know exactly which values are still being referenced. Unfortunately JavaScript does not have built-in functionality for this: *WeakMap*[6] has been designed to disallow this kind of checks. To support these features, we have a heap scanner that marks all heap objects reachable from running Haskell threads. Since all Haskell heap objects are just regular JavaScript objects, it's easy to inspect all values. Scanning the heap is an expensive operation, but fortunately does not need to be done very often, and can be temporarily disabled.

1.3 Concurrency in the browser

JavaScript does not have shared-memory multithreading. While it is possible to have true parallellism in JavaScript with Web Workers, their memory isolation makes them very inconvenient for implementing Haskell lightweight threads.

Therefore, GHCJS implements it own scheduler that runs in a single JavaScript thread. Since we share this thread with the user interface, which becomes unresponsive while our code is running, we have to be careful with long-running code and blocking operations.

A common pattern in JavaScript is to use callbacks, or continuations, for operations that produce a result later. This pattern is supported by the JavaScript runtime for some operations, for example:

```
jQuery.ajax('http://haskell.org/').done(f);
```

This code does an HTTP request (using the popular jQuery library) that retrieves the *haskell.org* page and calls *f* with the result. Another example is the *setTimeout* function, used below to let JavaScript call *g* after 100 milliseconds:

```
setTimeout(g, 100);
```

Both calls are asynchronous: The functions return immediately, the JavaScript runtime make sure that the continuation is called. The GHCJS runtime makes heavy use of this pattern, mostly implicitly, letting the scheduler yield to the browser, in order to let other code run and keep the user interface responsive. There is also direct support, through the foreign function interface:

```
foreign import javascript safe
  "jQuery.ajax($1).done($c)"
  ajax :: JSString -> IO JSString
```

This is an asynchronous foreign import. The current Haskell thread is suspended until *$c* is called from the foreign code. The GHCJS runtime continues to run other threads.

Whereas JavaScript code depending on *jQuery.ajax* results always has to be written in continuation passing style, Haskell code using the *ajax* FFI call does not look different from any other IO action. This makes exception handling much more convenient. We could use *System.Timeout.timeout* to deal with unresponsive servers for example:

```
main = timeout 500 (ajax "http://haskell.org")
         >>= \case
               Nothing -> putStrLn "no reponse"
               Just x  -> print x
```

On top of the scheduler, synchronization primitives like MVar and STM have been implemented.

1.4 Synchronous threads

While automatic handling of asynchronous actions in the GHCJS runtime is convenient most of the time, there are some situations where it gets in the way. A good example is event handling. Consider the following scenario:

```
myButton.on('click', function(event) {
  if(someCondition(event)) {
    event.stopPropagation();
  }
});
```

The code registers an event listener that fires when the button is clicked. Depending on *someCondition(event)*, further propagation of the event is stopped by calling *event.stopPropagation* before the listener returns.

If we wanted to convert the handler to a Haskell, we would have a problem: Any Haskell code might block, even pure code, for example when encountering a black hole from another thread. This makes it impossible to predict whether *event.stopPropagation* is called in time!

GHCJS has an alternative kind of thread: Synchronous threads, for which the runtime tries to run as much code as possible before returning, including temporarily switching to other threads to clear black holes. If it turns out to be impossible to run the thread to completion without blocking, a JavaScript exception is thrown. The thread is then either aborted, or continued asynchronously afterwards.

The example below creates a synchronous callback using *syncCallback1*. This callback, which is a JavaScript function that runs an IO action in a synchronous GHCJS thread, is then installed as an event handler for the HTML element *myButton*.

```
foreign import javascript unsafe
  "jQuery('#myButton').on('click', $1)"
  installMyButtonHandler :: JSFun (JSEvent -> IO ()) -> IO ()

handler :: JSEvent -> IO ()
handler _ = putStrLn "myButton clicked"

main = installByMuttonHandler =<<
          syncCallback1 False False handler
```

Asynchronous FFI and things like *threadDelay* are unsupported in synchronous threads.

2. Demonstration

The demonstration will explain the basics of the GHCJS runtime system, as described above, with examples of interaction with JavaScript through the FFI and running code in the browser.

References

[1] Haskell Wiki: The JavaScript Problem (`http://www.haskell.org/haskellwiki/The_JavaScript_Problem`)

[2] Jurriën Stutterheim - Improving the UHC JavaScript Backend, Utrecht 2012

[3] Fay: A proper subset of Haskell that compiles to JavaScript (`http://fay-lang.org`)

[4] Anton Ekblad - Towards a Declarative Web. MSc Thesis, Gothenburg, Sweden 2012

[5] Haskell Wiki: JMacro (`http://www.haskell.org/haskellwiki/Jmacro`)

[6] ECMAScript 6 specification draft (`http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`)