GHCJS

And how to use it

Artem Chirkin

GHCJS is a Haskell to JavaScript compiler

GHCJS is a Haskell to JavaScript compiler that uses the GHC API.

GHCJS supports many modern Haskell features, including:

- All type system extensions supported by GHC
- Lightweight preemptive threading with blackholes, MVar, STM, asynchronous exceptions
- Weak references, CAF deallocation, StableName, StablePtr
- Unboxed arrays, emulated pointers
- Integer support through JSBN, 32 and 64 bit signed and unsigned arithmetic (Word64, Int32 etc.)
- Cost-centres, stack traces
- Cabal support, GHCJS has its own package database

And some JavaScript-specific features:

- new JavaScriptFFI extension, with convenient import pattens, asynchronous FFI and a JSRef FFI type,
- synchronous and asynchronous threads.

Got this directly from README.md on https://github.com/ghcjs/ghcjs

Installation and HelloWorld

- Available on GHCJS github repository
  Compilation of the compiler and the base packages takes a while

  ```
  $ cabal install ./ghcjs # standard way of installing stuff
  $ ghcjs-boot  # install ghcjs-base packages into cabal .ghcjs repository
  ```

- For compilation use ghcjs
  or cabal --ghcjs flag
  or compiler: ghcjs in cabal.config of a sandbox

Compiled Hello World

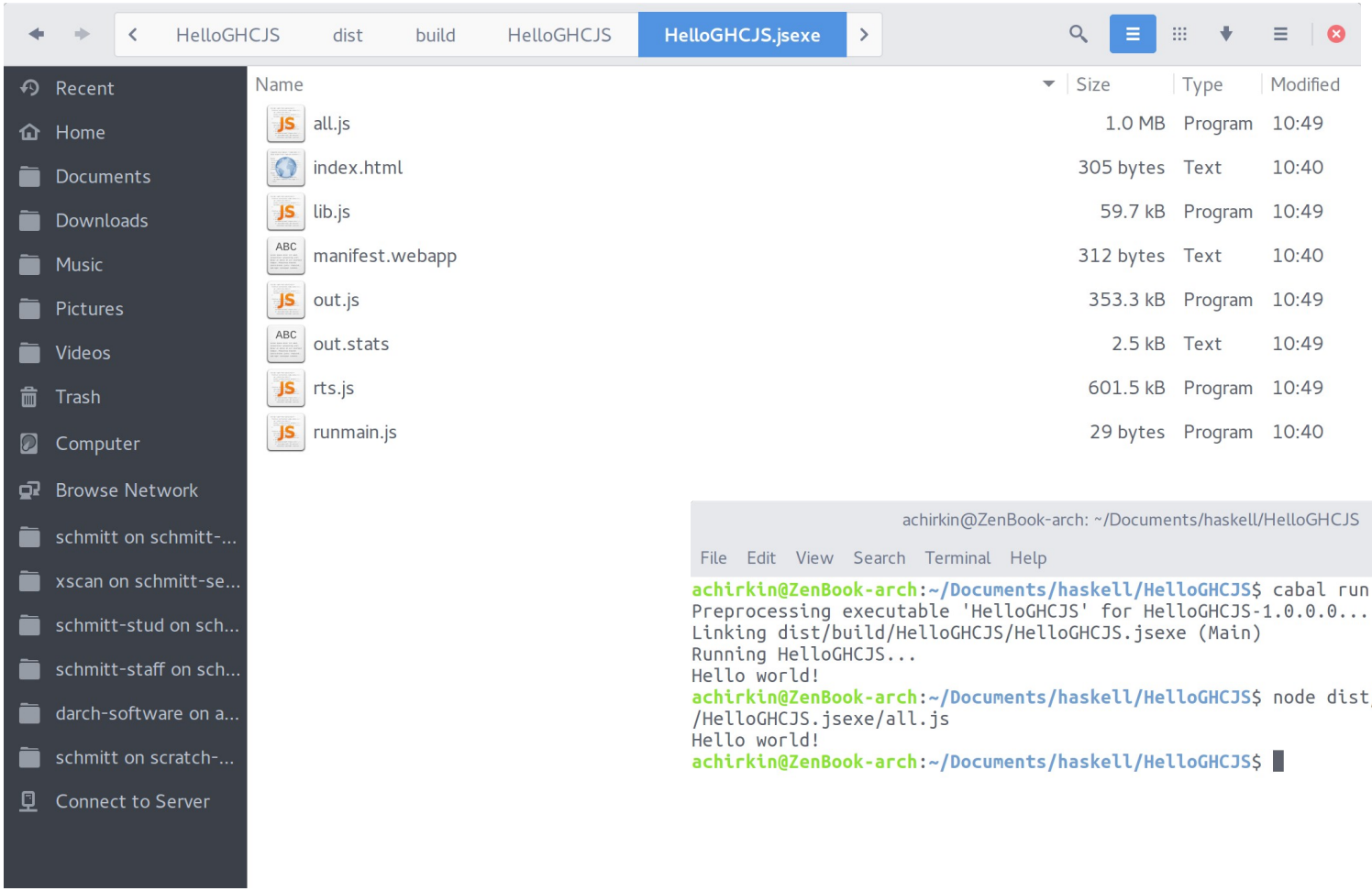
HelloGHCJS.jsexe/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <script language="javascript" src="rts.js"></script>
    <script language="javascript" src="lib.js"></script>
    <script language="javascript" src="out.js"></script>
  </head>
  <body>
  </body>
  <script language="javascript" src="runmain.js" defer></script>
</html>
```

On compilation ghcjs emits index.html test page and four js files:

runmain
+ lib (foreign (js) code – js-sources line in projec.cabal file)
+ out (our code)
+ rts

Compiled Hello World



Output sizes (out.stats)

HelloGHCJS – 2MB

```
code size summary per package:

base:                    278061
ghc-prim:                3321
ghcjs_HwzX8eXAwveCUOoAWSdVDg:8376
integer-gmp:             1997
main:                    839


code size per module:

base
    Control.Exception.Base:          2754
    ...
    System.Posix.Internals:          16161

ghc-prim
    GHC.CString:                     1876
    GHC.IntWord64:                   146
    GHC.Tuple:                       542
    GHC.Types:                       757

ghcjs_HwzX8eXAwveCUOoAWSdVDg
    GHCJS.Prim:                      7600
    GHCJS.Prim.Internal:             776

integer-gmp
    GHC.Integer.GMP.Prim:            172
    GHC.Integer.Type:                1825

main
    Main:                            839


packed metadata: 60743
```

ghcjs-modeler – 6MB

```
code size summary per package:

base:                    820728
...
fgeom_0evYSJPzEL1G09aAlbVeCu:1144733
geojs_3CbpcJY6jNjI2tVx96NtiU:360433
ghc-prim:                28063
main:                    1645078
...
vecto_EcjDyDlwiMg9atEfp9gJxH:2164


code size per module:

...

main
    Controllers.ElementResizing:     1339
    Controllers.GUIEvents:           1745
    Controllers.GeoJSONFileImport:   9290
    Controllers.LuciClient:          35407
    Controllers.Pointer:             146774
    GHCJS.Useful:                    14126
    Main:                            32851
    Program:                         15973
    ...
    Reactive:                        75721
    Services:                        780
    Services.Isovist:                161159
    Services.RadianceService:        9902
    SmallGL.Helpers:                 24559
    SmallGL.Shader:                  41356
    SmallGL.WritableVectors:         40912

...


packed metadata: 855934
```

Closure compiler is our friend

- The command I use for ghcjs-modeler:
- ```
  $ closure-compiler --language_in=ECMASCRIPT5 dist/build/ghcjs-modeler/ghcjs-modeler.jsexe/all.js web/misc.js web/luciConnect.js web/faultylabs.md5.js --compilation_level=ADVANCED_OPTIMIZATIONS > web/modeler.js
  ```
- This combines libraries (~200KB) and ghcjs output (~6MB) into 2MB .js file.
  ```

ASM.js and WebAssemblies are not here

- GHCJS is not going to support ASM.js

luite commented on Mar 27, 2013 — Owner

not really unless the manually managed heap branch is resurrected again.

asm.js is really geared towards running traditional imperative code, functional code like ghcjs produces relies much more on passing around functions with associated data. it can possibly be fixed by storing all closure entry points in a large array and using integers to emulate function pointers, but that adds an extra layer of indirection to function calls

https://github.com/ghcjs/ghcjs/issues/53

- And the same problem with WebAssembly

luite commented on Jun 18 — Owner

Unfortunately so far it appears to have the same limitations. No access to the kind of objects we use for the heap.

The new codegen tracks type information more closely, so the typed int32/64 things could actually benefit, but without a lightweight way to access JS data structures it would require a massive rewrite to support it.

https://github.com/ghcjs/ghcjs/issues/359

luite commented on Jun 18 — Owner

haskell heap objects are stored as JS objects: `var heapObj = { f: someFunction, d1: data1, d2: data2, m: metadata }`. I don't think it's possible to encode this in the binary AST.

Threading and web workers

- GHCJS implements its own lightweight threading system which runs in a single js thread.
- In current version of ghcjs-base WebWorkers are not supported. I suppose, the main problem is that WebWorker threads are isolated, thus we need another copy of RTS running on WebWorker to make it run Haskell code.
- There is a branch of ghcjs on github called `improved-base`. It has WebWorkers, but I have not tried it yet.

Foreign calls & GHCJS.Foreign

The principal type here is `JSRef`, which is a lifted type that contains a JavaScript reference. The `JSRef` type is parameterized with one phantom type, and `GHCJS.Types` defines several type synonyms for specific variants.

The code in this module makes no assumptions about '`JSRef a`' types. Operations that can result in a JS exception that can kill a Haskell thread are marked unsafe (for example if the `JSRef` contains a `null` or `undefined` value). There are safe variants where the JS exception is propagated as a Haskell exception, so that it can be handled on the Haskell side.

For more specific types, like `JSArray` or `JSBool`, the code assumes that the contents of the `JSRef` actually is a JavaScript array or bool value. If it contains an unexpected value, the code can result in exceptions that kill the Haskell thread, even for functions not marked unsafe.

The code makes use of `foreign import javascript`, enabled with the `JavaScriptFFI` extension, available since GHC 7.8. There are three different safety levels:

- `unsafe`: The imported code is run directly. returning an incorrectly typed value leads to undefined behaviour. JavaScript exceptions in the foreign code kill the Haskell thread.
- `safe`: Returned values are replaced with a default value if they have the wrong type. JavaScript exceptions are caught and propagated as Haskell exceptions (JSException), so they can be handled with the standard `Control.Exception` machinery.
- `interruptible`: The import is asynchronous. The calling Haskell thread sleeps until the foreign code calls the `` `$c` `` JavaScript function with the result. The thread is in interruptible state while blocked, so it can receive asynchronous exceptions.

Unlike the FFI for native code, it's safe to call back into Haskell (`h$run`, `h$runSync`) from foreign code in any of the safety levels. Since JavaScript is single threaded, no Haskell threads can run while the foreign code is running.

Warning! “JSRef a” is changed to “JSVal”, with some consequences. See the last slide.

Taken from GHCJS.Foreign

Foreign calls & GHCJS.Foreign

Many examples are in GHCJS.Useful module of ghcjs-modeler

https://github.com/achirkin/ghcjs-modeler/blob/master/src/GHCJS/Useful.hs

Handle JS objects and JSON

JSON that comes from outside (via JavaScript) is always an external js object, so it has type "`JSRef a`" in Haskell.
But we would like to use our beloved aeson, right?

```
-- | These declarations are in GHCJS.Marshal
class FromJSRef a where
    fromJSRef :: JSRef a -> IO (Maybe a)

class ToJSRef a where
    toJSRef :: a -> IO (JSRef a)

toJSRef_aeson :: ToJSON a => a -> IO (JSRef a)

-- | Easy to add JSON import on our own
fromJSRef_aeson :: FromJSON a => JSRef a -> IO (Maybe a)
fromJSRef_aeson = liftM (>>= f . fromJSON) . fromJSRef . castRef
    where f (Error _) = Nothing
          f (Success x) = Just x
```

Warning! This is an obsolete way for the new release of ghcjs. See the last slide.

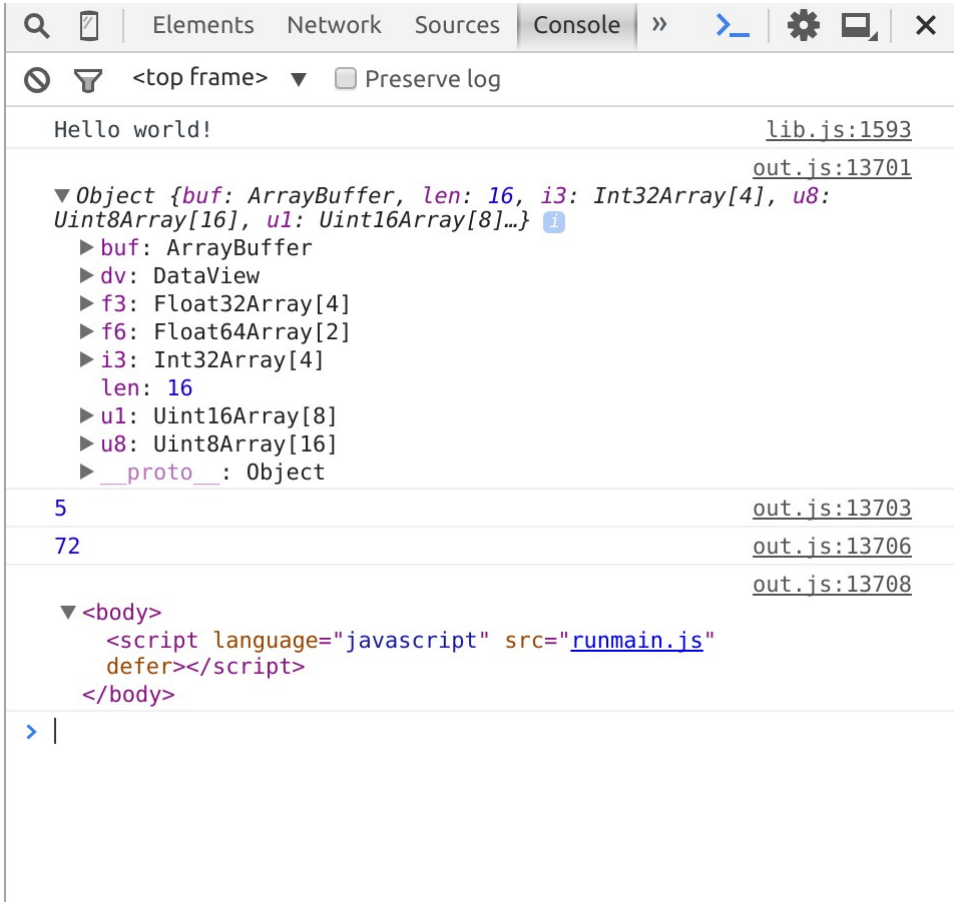
More on JSRef and ByteBuffers

- Dirty hack:

  By doing `unsafeCoerce` to `GHC.Exts.Any` we can get an insight how the objects in GHCJS Haskell are mapped onto JavaScript.
  Note! It is not a conversion of types; it would be just the same as trying to print Ptr () in standard Haskell. The dirty trick is that we can always print anything in JS. So we do print ghcjs heap objects!

```
import GHC.Exts as Exts
-- | In JS we can print just anything
printAnything :: a -> IO ()
printAnything a = printAny (unsafeCoerce a :: Exts.Any)

foreign import javascript unsafe "console.log($1)"
    printAny :: Exts.Any -> IO ()
```



WebGL (updated slide)

Look into ghcjs-webgl source code

Warning!

ghcjs-webgl now is adapted to a new ghcjs-base. New ghcjs does much better integration of js types.

https://github.com/achirkin/ghcjs-webgl/blob/master/src/GHCJS/WebGL/Types.hs

The new type for JS values: `JSVal` instead of `JSRef a` .
The new preferred way of wrapping JS types is as follows:

```
newtype WebGLRenderingContext = WebGLRenderingContext JSVal
instance IsJSVal WebGLRenderingContext
```

This allows to `Data.coerce` it to `JSVal` and back, or use the newtyped values directly in javascript imports!

```
foreign import javascript unsafe "$1.activeTexture($2)"
    activeTexture :: WebGLRenderingContext -> GLenum -> IO ()
```