

Announcement



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Announcing *DHD* \gg *UHac*

- ▶ Announcing *DHD* \gg *UHac*
 - ▶ *Dutch HUG Day*: several highly interesting presentations
 - ▶ *UHac*: a weekend-long Haskell hackathon
- ▶ Date: 20-22 April 2012
- ▶ Location: Utrecht
 - ▶ *DHD* is hosted at and sponsored by Ordina
 - ▶ *UHac* is sponsored by Utrecht University
- ▶ Looking for speakers for *DHD*
- ▶ More information:
http://www.haskell.org/haskellwiki/DHD_UHac



Improving the UHC JavaScript backend

Jurriën Stutterheim

6 January, 2012



Current situation

- ▶ Nobody¹ really likes JavaScript programming
- ▶ Yet, we are forced to by evil browser vendors ☹
- ▶ Luckily, the Utrecht Haskell Compiler (UHC) offers us an alternative!



¹at least, no Haskell programmer that I know

The UHC JavaScript backend

- ▶ Write Haskell, compile it to JavaScript
- ▶ First implementation by Atze Dijkstra
- ▶ Enjoy many of Haskell's desirable features in the generated JavaScript code
 - ▶ Type-safety
 - ▶ Lazy evaluation
 - ▶ Partial function application



Other (potential) benefits

- ▶ Libraries can be used on both client and server
- ▶ Move (possibly complex) business logic from the server to the client, eliminating XHRs, improving responsiveness
- ▶ Use QuickCheck for (indirectly) testing JavaScript code
- ▶ Or go even further: use Coq to generate proved Haskell code, which is then compile to JavaScript



Inner workings: main ingredients

Model functions, function application and explicit evaluation in JavaScript

- ▶ Functions: `new _F_(function (..) {..})`
- ▶ Function application: `new _A_(new _F_(..), [...])`
- ▶ Evaluation: `_e_(..)`



(Simplified) Example

The Haskell function `add x y z = x + y + z` is roughly compiled to:

```
add = new _F_(function (x, y, z) { return x + y + z;});
```

which can be (potentially partially) applied to a list of arguments. The JavaScript representation of `add 3 4 5`:

```
app345 = new _A_(add, [3, 4, 5]);
```

We can get our answer by evaluating the expression:

```
answer = _e_(app345);
```

After which the value of `answer` is 12



Importing JavaScript function

We can import JavaScript functions into Haskell using the Foreign Function Interface (FFI). Suppose we have the following JavaScript function:

```
mySum = function(x, y) { return x + y; }
```

Importing it in Haskell:

```
foreign import jscript "mySum(%1, %2)"  
mySum :: Int → Int → Int
```

After which we can use it like any other Haskell function:

```
print (mySum 2 3)
```

prints 5



Exporting a Haskell function

Conversely, we can export a Haskell function to JavaScript. Suppose we write and export the following Haskell function:

```
mySum :: Int → Int → Int  
mySum x y = x + y  
foreign export javascript "mySum"  
mySum :: Int → Int → Int
```

It would generate the following JavaScript code:

```
mySum = function(x, y) {  
  return _e_(new _A_(haskMySum, [x, y]));  
}
```

The Haskell function can now be used by existing JavaScript libraries.



New in the UHC JS backend

We could already do the things on the previous slides last year. This year we can do more!

- ▶ Runtime object creation, querying and manipulation
- ▶ Runtime conversion from Haskell datatypes to JS objects
- ▶ Support for using Haskell functions as JS callbacks with *wrapper* imports
- ▶ Support for dynamically importing JS functions as Haskell functions with *dynamic* imports



Dealing with JavaScript objects

- ▶ JavaScript supports prototype-based OOP
- ▶ Haskell is purely functional and has no notion of objects
- ▶ How do we interact with JavaScript objects from Haskell?
- ▶ We want to ...
 - ▶ ... pass objects around
 - ▶ ... get the data stored inside objects
 - ▶ ... modify objects
 - ▶ ... create new objects
 - ▶ ... do all of this without doing a lot of extra manual work



Representing objects

- ▶ The only way to represent objects in Haskell is as an opaque pointer type
- ▶ For JavaScript objects, we define this type as `data JSPtr a`
- ▶ The only way to obtain values of this type is via the FFI, since it does not have any constructors

We can obtain objects from the FFI, but how do we use the FFI to create, query and manipulate objects?



New compiler primitives

- ▶ One can define plain JavaScript functions that wrap around the normal interface for creating, querying and manipulating objects
- ▶ A set of such functions has been added to UHC's set of primitive JavaScript functions
- ▶ These functions can then be imported into Haskell using the *prim* FFI
- ▶ Result: object interaction with a functional flavour



Primitives: creating objects

Creating a new constructor

$$\text{primMkCtor} :: \text{JSString} \rightarrow \text{IO } ()$$

Instantiate an object of a given constructor, creating the constructor if needed

$$\text{primMkObj} :: \text{JSString} \rightarrow \text{IO } (\text{JSPtr } a)$$

Instantiate an anonymous object (`{}` in JavaScript)

$$\text{primMkAnonObj} :: \text{IO } (\text{JSPtr } a)$$


Primitives: querying and modifying objects

We are losing a lot of type-safety here...

$\text{primGetAttr} :: \text{JSString} \rightarrow \text{JSPtr } b \rightarrow \text{IO } a$

$\text{primSetAttr} :: \text{JSString} \rightarrow a \rightarrow \text{JSPtr } b$
 $\rightarrow \text{IO } (\text{JSPtr } b)$

$\text{primModAttr} :: \text{JSString} \rightarrow (a \rightarrow b)$
 $\rightarrow \text{JSPtr } c \rightarrow \text{IO } (\text{JSPtr } c)$

Similar primitives are available for dealing with prototype attributes



Pure variants

Pure changes can be simulated by first cloning an object and then modifying the clone

$$\text{primClone} :: \text{JSPtr } a \rightarrow \text{JSPtr } a$$

Which allows us to define the following (albeit inefficient) functions, without getting stuck in *IO*

$$\begin{aligned} \text{primPureSetAttr} &:: \text{JSString} \rightarrow a \rightarrow \text{JSPtr } b \\ &\rightarrow \text{JSPtr } b \end{aligned}$$
$$\begin{aligned} \text{primPureModAttr} &:: \text{JSString} \rightarrow (a \rightarrow b) \\ &\rightarrow \text{JSPtr } c \rightarrow \text{JSPtr } c \end{aligned}$$


Haskell *uhc-jscript* library

We offer a standard library that imports these primitives and wraps them in a slightly nicer API:

```
mkObj   :: String → IO (JSPtr a)  
getAttr :: String → JSPtr b → IO a  
setAttr  :: String → a → JSPtr b → IO (JSPtr b)  
modAttr :: String → (a → b) → JSPtr c  
          → IO (JSPtr c)
```

The library also contains ECMA APIs, HTML APIs, jQuery APIs and more



Creating objects, the laborious way

```
main :: IO ()
main = do
  b ← mkObj "Book"
  setAttr "author" "Lipovaca" b
  setAttr "title" "LYAH" b
  setAttr "pages" 400 b
  setAttr ...
  ...
```

We do not want to do this for all our JavaScript objects!



JavaScript objects and Haskell datatypes

Observation: Haskell constructors are very similar to JavaScript objects

```
book  
= Book  
{ author = toJSString "Lipovaca"  
  , title   = toJSString "LYAH"  
  , pages  = 400 }
```

```
book  
=  
{ author : "Lipovaca"  
  , title   : "LYAH"  
  , pages  : 400 }
```



Automatic conversion

UHC has a new special object wrapper import

```
foreign import jscripT "{}"  
  toObj :: a → IO (JSPtr b)
```

Now we can convert datatypes to JavaScript objects at runtime

```
main = do  
  let b' = book {pages = pages book + 1}  
      b ← toObj b'  
      p ← getAttr "pages" b  
      print p -- Prints 401
```



Dynamic and wrapper imports

- ▶ Runtime constructs for interfacing the JavaScript and Haskell worlds
- ▶ Dynamic: convert a pointer to a JavaScript function to a Haskell function
 - ▶ For dealing with higher-order JavaScript functions that return a JavaScript function
 - ▶ $FunPtr\ ft \rightarrow ft$
- ▶ Wrapper: convert a Haskell function to a JavaScript function pointer
 - ▶ For allowing plain JavaScript to call Haskell functions (e.g. use a Haskell function as callback in a JavaScript library)
 - ▶ The Haskell 2010 Language Report defines the wrapper type as $ft \rightarrow IO\ (FunPtr\ ft)$, but the current implementation in UHC is restricted to monadic types:
 $m\ ft \rightarrow IO\ (FunPtr\ (m\ ft))$



Conclusion

- ▶ With the work presented here we can now (almost) write complete JavaScript applications using Haskell
- ▶ Although we currently still require some workarounds here and there
- ▶ Especially wrt. dealing with `this`
- ▶ Help with hacking and ideas are most welcome
- ▶ In addition to type-safety, this provides us some other interesting options



Grand finale

A demo of Alessandro Vermeulen's effort to port one of my applications' JavaScript to Haskell.

