Haskell and WebAssembly

An Introduction based on Asterius

Nicolas Karrer
OST Eastern Switzerland University of Applied Sciences
MSE Seminar "Programming Languages"
Supervisor: Farhad Mehta
Spring 2022
Switzerland

Abstract

When writing code, different programming languages offer various approaches to solve a problem. Some of those problems might be elegantly solved in one programming language, but challenging in a different one. When it comes to the Web, interactive client-applications rely only on JavaScript. With the introduction of WebAssembly, this limitation has fallen. This article gives some thoughts, why it can be useful to write applications for the Web in another language than JavaScript, and it provides an introduction, on how to compile Haskell code to WebAssembly with the Haskell-to-WebAssembly-Compiler Asterius. It shows an approach how to compile a full Haskell application, as well as making single Haskell functions available in JavaScript. It can be challenging, when it comes to the interaction between a strictly typed language, like Haskell, and the dynamically typed language JavaScript. Based on examples, the article gives insight into this challenge, and its solution provided by Asterius.

Keywords: WebAssembly, Asterius, JavaScript, Haskell

1 Introduction

The intention of this article is to reflect on the insights gained from trying out the Haskell-to-WebAssembly-Compiler Asterius. The Article aims to be an aggregation and summary of the, in my view, most important take aways for beginners, provided by the several blogposts and tutorials of the creators of Asterius, as some of the information is distributed. In this article I will discuss two cases Asterius provides to compile Haskell to WebAssembly:

- Section 5: Compiling a Haskell Application to WebAssembly and run it in the browser. In this case, the main function is used as the entry point. It provides the possibility to run Haskell-Applications, targeting the browser or the NodeJS environment.
- Section 6: Compiling a single function or a set of functions to WebAssembly. In my opinion, this is an interesting case: Haskell code could be used as library code. It would be possible to make use of some strengths of the Haskell Programming Language within a JavaScript application.

2 Background

Asterius is a compiler based on the Glasgow Haskell Compiler (GHC) for Haskell[1], targeting WebAssembly. It compiles Haskell Code to WebAssembly, and it provides infrastructural code to run the resulting WebAssembly code in the browser and the NodeJS environment. It contains library code to help with type marshalling on the Haskell and the JavaScript side. Tweag I/O[2] maintains Asterius. They provide introductory tutorials and examples[3] as well as more in-depth documentation[4]

3 Motivation

While server-side applications for the Web are not restricted to a single programming-language, client-side applications on the other hand rely on JavaScript. Writing meaningful, interactive applications for the browser therefore requires developers to learn JavaScript. There were approaches to run applications written in other languages for the browser, for example Java or Flash, but they were not adopted broadly or abandoned by the browser providers for a multitude of reasons. [Ora] [Goo17]

Other approaches like emscripten or ASM.js used JavaScript or a subset of JavaScript as a compilation target. But as JavaScript is not designed as a compilation target, limitations persist. WebAssembly was introduced as a low-level bytecode, especially design to work as a compilation target. It was claimed to be the first compilation target which is safe, fast, portable, and compact. [HRS+17]

Browsers are adopting features which give web applications the possibility to act like a native app. This includes the ability to run without internet access or accessing the device's cameras and position. This means more potential use cases for web applications arise. An example of such a use case would be a bar code scanner for e-commerce and shopping apps. It requires knowledge in image recognition, which, from my experience working in the field of web development, seems not to be wide spread in the Web community.

[1] https://www.haskell.org/ghc/
[2] https://www.tweag.io/
[3] https://www.tweag.io/blog/tags/asterius
[4] https://asterius.netlify.app/

Therefore, giving more developers access to not only the server-side, but also to the client-side of the Web can be beneficial. It has the potential to increase the amount of good quality web applications, especially when it comes to attributes of special knowledge, performance, or reliability.

As Haskell is a pure functional language, is statically typed and provides a lot of language features to solve problems elegantly, it is an interesting language for JavaScript programmers to dive into and sharpen their knowledge. They could improve their skills in functional programming, which JavaScript is also capable of. They might take advantage of performance improvements for intensive computation, or they could use the language features of Haskell to solve programming problems more adequately and reliable than it would be possible with JavaScript.

4 Prerequisite

For interested readers, wanting to try out Asterius, the provider recommend, using the docker-image[5] to compile the Haskell code to WebAssembly.

The commands used in this article assume that they are run on the command line interface (CLI) inside the Docker container. The necessary docker image can be pulled from Docker Hub with the command from Listing 1. Listing 2 shows how the Docker container is started from the current directory.

```
1  docker pull terrorjack/asterius
```

Listing 1. The command to pull the Asterius Docker image.

```
1  docker run -it --rm -v .:/workspace -w /workspace
       terrorjack/asterius
```

Listing 2. The command to start the Asterius Docker container.

As most browsers do not allow file access to the local file system for security reasons, at least a simple web server is necessary to run the WebAssembly code in the browser. To test and run code from the examples in this article, the `http-server`[6] package from NPM (Node package manager) was used.

5 Use case 1: Running a Haskell application in the browser

Compiling Haskell code to be run in the browser for a simple example is straight forward. The code in Listing 3 doubles a given integer and prints the output in the browser console:[7]. The command from Listing 4 compiles the Haskell code to WebAssembly.

```
1  import Control.Monad
2  import System.IO
3
4  double :: Int -> Int
5  double x = 2 * x
6
7  main :: IO
8  main = print $ double 3
```

Listing 3. The content of the file double.hs.

```
1  ahc-link --browser --input-hs double.hs
```

Listing 4. The command to compile double.hs to WebAssembly.

The compilation results in a set of files. Those files are necessary to run the code in the browser. The most interesting ones are *double.wasm*, *double.html* and *double.mjs*. While *double.wasm* represents the WebAssembly bytecode of the program. The files *double.html* and *double.mjs* contain generated code to help running the resulting WebAssembly code in the browser.

The result of the program execution can be seen, when serving *double.html* to the browser via a web server. With in *double.html* the file *double.mjs* is loaded. *double.mjs* loads then the WebAssembly code from the web server and executes it asynchronously by calling the exported `main` function. The resulting "6" is printed to the console, visible in the browser's developer tools.

In of itself this code is not very useful but shows some initial mechanics how to use the Asterius compiler and what it produces.

Tweag I/O provides two, more in depth tutorials which are worth mentioning for this use case:

The Hilbert Tutorial [SHI19]: This tutorial shows how to print a graphic to the browser, using the Scalable Vector Graphics (SVG) image format. It uses the Haskell package manager Cabal to import already available packages from the Haskell ecosystem for generating SVG-graphics. It uses `import javascript` to render the resulting graphic in the browser. This is possible because Asterius provides an adaption of Haskell's JavaScript Foreign Function Interface (JSFFI)[Sha18b]. With `import javascript` a single JavaScript expression can be executed from within Haskell. The opposing `export javascript` will be discussed in section 6. Both methods from the JSFFI are used when marshalling types.

It is worth mentioning, that in a real-world application it would be a better approach to append the graphic not directly to the body, but into a predefined existing element. This allows a controlled positioning of the resulting application inside an existing webpage. The adaption of the necessary changes to the tutorial code are shown in Listing 5 and Listing 6.

```
1  foreign import javascript
2    "(() => {                                          \
3    \   const d = document.querySelector('#show-
         hilbert-here'); \
4    \   d.innerHTML = $1; \
5    \ })()"
```

[5] https://hub.docker.com/r/terrorjack/asterius
[6] https://www.npmjs.com/package/http-server
[7] https://balsamiq.com/support/faqs/browserconsole/

```
6     showSVG :: JSString -> IO ()
```

Listing 5. The adaption of the Hilbert tutorial to target a specific HTML element by its ID.

```
1  <body>
2     <p>Shows the Hilbert output</p>
3
4     <div id="show-hilbert-here">
5        <!-- Resulting SVG will be displayed here -->
6     </div>
7
8     <script type="module" src="Hilbert.mjs"></script>
9  </body>
```

Listing 6. The adaption of the Hilbert tutorial, providing the specific HTML element.

Note: As Asterius will overwrite the *Hilbert.html* file upon a new compilation, I recommend extracting the code into a new *index.html* file and adapt this.

The Todo-MVC Example [Sha18a]: This tutorial goes deeper into details on how to write an Application in Haskell and compile it to WebAssembly. It introduces new concepts such as type marshalling and the element abstraction. Type marshalling is necessary to work with types across the language barrier of Haskell and JavaScript. A more detailed explanation is provided in section 7.

In the previous tutorial, the manipulation of the HTML Document Object Model (DOM) was done via invoking JavaScript from Haskell. The Element abstraction by Asterius provides a Haskell native functionality to interact with the DOM.

This section has shown the necessary steps to compile Haskell applications to WebAssembly with the main function, and run the resulting WebAssembly code in the browser. With the provided Docker image, there is no need for an extensive setup procedure. All necessary tools are already installed. Interacting with the browser from Haskell can either be done with the element abstraction or by calling JavaScript from Haskell. The browser's development tool console is treated as the output device for print, which one would expect. Some manual work might be necessary to extract the resulting code, as Asterius overwrites already existing files, even if they are modified. Some adaptions to this process would be helpful to prevent unexpected behaviour.

6 Use case 2: Make Haskell functions available in JavaScript

Some programming problems can be solved very elegantly in Haskell. It is pure functional and provides features like type safety and laziness. In addition, it provides some syntactical benefits like list comprehension and pattern matching.

While JavaScript does a good job providing interaction to a web application, it lacks features that are available in Haskell. Missing those features can have an impact on performance and reliability of an application written in JavaScript.

One assumes it might be a good idea to delegate certain critical tasks and problems to a more appropriate programming language. Therefore, an interesting case, when working with Haskell and WebAssembly, is to make single Haskell functions available in JavaScript. Asterius does provide the functionality to compile single Haskell functions to WebAssembly and make them usable in JavaScript.

This section and section 7 focus on the necessary steps and challenges that arise, when making Haskell functions available in JavaScript.

Revisiting the example double from the previous section, it is necessary to do some adaptions to the Haskell code, the compilation command and to the resulting generated JavaScript code.

For the function `doubleInt` to be available in JavaScript, the Haskell code has to fulfil the two additional requirements, shown in Listing 7:

1. The code must be defined inside a module definition.
2. The function must be explicitly exported to JavaScript.

```
1  module ExportedHSFunction where
2
3  foreign export javascript doubleInt :: Int -> Int
4  shouldBeExportedInt x = 2 * x
```

Listing 7. The content of file doubleInt.hs

Note: if the first requirement is not fulfilled, a compilation error occurs, because of the missing main function. If the second requirement is not fulfilled, the code will be compiled. But the function will not be available in JavaScript.

The compilation command has to fulfil two additional requirements as well, as shown in Listing 8:

1. The code must be compiled with the no-main parameter.
2. The code must be compiled with the `-export-function` parameter.

```
1  ahc-link --browser --input-hs doubleInt.hs --no-main
       --export-function doubleInt
```

Listing 8. The command to compile doubleInt.hs to WebAssembly

The resulting code in the *doubleInt.mjs* file requires adaption as well. Asterius generates the file still calling the `main` function, instead of the new `doubleInt` function. As this change is done in an auto-generated file, which will be overwritten upon a subsequent compilation, it is recommended creating a copy of the file and adapt the resulting JavaScript code as shown in Listing 9.

Lines 1 to 5 in Listing 9 is the JavaScript code generated by Asterius. It takes care of loading and instantiating the WebAssembly Code. No adaption is necessary here.

Lines 6 to 9 in Listing 9 is the adapted code that calls the target function `doubleInt`. Note that WebAssembly code must be called asynchronously in JavaScript. In this example,

it is done using a Promise Object of JavaScript. These objects are a way of handling callbacks in JavaScript. [4]

```
1  import * as rts from "./rts.mjs";
2  import module from "./export-function.wasm.mjs";
3  import req from "./export-function.req.mjs";
4
5  module.then(m => rts.newAsteriusInstance(Object.
       assign(req, { module: m }))).then(i => {
6      i.exports.doubleInt(3)
7          .then(result => {
8              console.log(result);
9          })
10 });
```

Listing 9. The function `doubleInt` is called asynchronously in JavaScript

Opening the file resulting *doubleInt.html* in the Browser will have the same result as the previous example. It will print the number "6" to the developer console.

Some specialities regarding the type are notable for this example as well: as `Int` is a shared type between JavaScript and Haskell, it can be use it as it is. The code will compile and run without any marshalling necessary on the Haskell side. There are examples in section 7 where this is not possible.

Because JavaScript is dynamically typed, some potentially surprising calls of `doubleInt` are possible:

- `i.exports.doubleInt(0.2)` will result in `0`, because the number will be converted to an Int with value 0.
- `i.exports.doubleInt('c')` will result in a runtime error from WebAssembly, which is correct. In plain JavaScript, calling `2 * 'c'` would result in the "Not a Number" (`NaN`) bottom value.

It is possible to use further Haskell types out-of-the-box when compiling Haskell to WebAssembly. Besides `Int`, tests with `Char`, `Bool` and `Double` were successful.

One exception arising from the tests is the type `Float`: a function with parameter or return type `Float` will be compiled, but when calling the resulting function in JavaScript, it will result in a runtime error. My assumption is, that a function is missing in the generated JavaScript code to handle this type correctly.

For further reading, Tweag I/O provides a tutorial for making Haskell functions available.[8] It gives some background information about the challenges of calling Haskell functions from JavaScript. For example, if the Haskell environment is not yet initialized.

This section has shown, that exporting single functions or a set of functions is possible with Asterius. It is possible to use primitive types like Int, Char, Bool, Double without any additional conversion necessary. Some convenience regarding the resulting code is missing for developers, as the code generation still generates code calling the non-existent `main` function. So the generated code needs adaptions, but is overwritten upon subsequent compilations.

7 Marshalling Types

While we can use any type provided by Haskell inside the Haskell part of the application, even custom types. The types for the interaction across Haskell and JavaScript are restricted. As mentioned in section 6 primitive types shared between Haskell and JavaScript can be used directly as argument type and return type.

In JavaScript, there is a set of more types, which are possible to use and in Haskell there is the possibility to define custom types.

To be able to pass such types across the language barrier, they have to be marshalled. This section describes the process of type marshalling for the most important JavaScript types: `String`, `Array`, `Function` and `Object`. For those types Asterius provides an abstraction which can be used in the function definitions in Haskell: `JSVal`, `JSString`, `JSArray`, JSFunction and `JSObject`.

JSVal: This is the basic type and stands for an opaque value existing In JavaScript. The other types are `newtypes`[9] of `JSVal`[I/O22].

JSString: While a string in JavaScript is a primitive type[10]. In Haskell, a string is a list of `Char`. But as both languages share the primitive type `Char`, it is possible to marshal a Haskell `[Char]` into `JSString` and back. The Listing 10 shows the process of marshalling strings with the help of the functions `fromJSString` and `toJSString` provided by Asterius[11].

In both cases, each string representation is marshalled character by character. To get a better insight of the details of the marshalling process, Listing 10 provides the definition of `fromJSString` and `toJSString` as well. It is important to note, that `codePointAt` and `fromCodePoint` are both methods of the JavaScript string type. The expression `$1` is a reference to the parameter, passed to the function.

Calling the Haskell function `js_string_tochar "Hello World" 0` translates to the JavaScript call

`"Hello World".codePointAt(0)` and results in the character `H`. [con20b] [con20a]

```
1  -- to apply reverse to a string, the given JSString
       has to be marshalled to a Haskell String
2  foreign export javascript reverseString :: JSString
       -> JSString
3  reverseString x = toJSString (reverse (fromJSString
       x))
4
5  -- these functions are provided by the Asterius.
       Types module
6  toJSString :: String -> JSVal
```

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[8] https://www.tweag.io/blog/2018-09-12-asterius-ffi/

[9] https://wiki.haskell.org/Newtype

[10] https://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/

[11] https://github.com/tweag/asterius/blob/master/asterius/test/jsffi/AsteriusPrim.hs

```
7  toJSString =
8    foldl' (\s c -> js_concat s (js_string_fromchar c)
         ) js_string_empty
9
10 fromJSString :: JSVal -> String
11 fromJSString s = [js_string_tochar s i | i <- [0 ..
       js_length s - 1]]
12
13 foreign import javascript "\"\"" js_string_empty ::
       JSVal
14
15 foreign import javascript "$1.concat($2)"
16   js_concat :: JSVal -> JSVal -> JSVal
17
18 foreign import javascript "String.fromCodePoint($1)"
19   js_string_fromchar :: Char -> JSVal
20
21 foreign import javascript "$1.codePointAt($2)"
22   js_string_tochar :: JSVal -> Int -> Char
```

Listing 10. Marshalling a `JSString` to a Haskell String and reversing it.

JSArray: Marshalling JavaScript arrays and Haskell lists is a bit more challenging. This is because, like strings, JavaScript arrays and Haskell lists do not translate one to one. However, the necessary procedure to marshal these types is similar to the procedure for strings. First, the array has to be translated into a list of `JSVal` or one of its `newtype`. Second each entry has to be marshalled in to the specific type. The Listing 11 shows the according functions provided by the module `AsteriusPrim`. This article describes a more detailed example of the procedure in section 8.

Note: Both one advantage and one disadvantage for this case arise from the dynamic typing of JavaScript: The advantage is, like for `JSString` the function `js_concat` is used to append an element to the array. Both JavaScript types String and Array contain the method `concat`.[con20c] Because of the dynamic typing of JavaScript, the same Haskell function can be use for marshalling both types.

As for the disadvantage: JavaScript arrays can contain elements of any type (Not to be confused with Haskell's `Any` type). For example, a JavaScript array can look like this: `[1, 2, 3, "four"]`. If such an array is passed to the Haskell code, it will cause a runtime error during the marshalling process.

```
1  toJSArray :: [JSVal] -> JSVal
2  toJSArray = foldl' js_concat js_array_empty
3
4  fromJSArray :: JSVal -> [JSVal]
5  fromJSArray arr = [js_index_by_int arr i | i <- [0
       .. js_length arr - 1]]
6
7  foreign import javascript "$1.concat($2)"
8    js_concat :: JSVal -> JSVal -> JSVal
9
10 foreign import javascript "[]" js_array_empty ::
       JSVal
11
12 foreign import javascript "$1[$2]" js_index_by_int
       :: JSVal -> Int -> JSVal
```

Listing 11. Converting a `JSString` to a Haskell string and reversing it.

JSObject: Asterius does not provide functions, besides creating an empty JavaScript object, to marshal `JSObjects`. But once understood how the JavaScript Foreign Interface works, the process is rather straight forward and is best explained by example.

Listing 12 defines a Haskell record `Person` with its properties `name :: String` and `age :: Int`[Dev22].

```
1  -- defining the Person record
2  data Person = Person {age :: Int, name :: String}
       deriving (Show, Eq)
3
4  setAge :: Int -> Person -> Person
5  setAge newAge person = person {age = newAge}
```

Listing 12. Defining a record `Person` in Haskell with properties `name` and `age`

Listing 13 shows the process of marshalling the JavaScript object into a `Person` record. For this, it is necessary to access the single attributes of the object and marshal it into the correct Haskell type. As age is of basic type Int, this can be done directly with the `accessAge` function. The attribute name has to be marshalled from `JSString` to `String` first.

Note: both object attributes are accessed with a default value (`$1.name || 'J. Doe'`). This prevents runtime errors if the attribute on the JavaScript object `undefined`. For example, when the attribute is missing or misspelled.

```
1  -- Marshal from JavaScript input
2  convertObjectToPerson :: JSObject -> Person
3  convertObjectToPerson obj = Person {age = age',
       name = name'}
4      where
5          age' = accessAge obj
6          name' = (fromJSString . accessName) obj
7
8  foreign import javascript "$1.name || 'J. Doe'"
       accessName :: JSObject -> JSString
9  foreign import javascript "$1.age || 0" accessAge ::
        JSObject -> Int
```

Listing 13. Receiving a JavaScript object and marshal it into the `Person` record

Listing 14 shows the process of marshalling the `Person` record back into a JavaScript object. Note that the numbering of the variables $1, $2 … $n in function `jsPerson` is defined by the order of its parameters.

```
1  -- Marshal to JavaScript outputs
2  convertPersonToObject :: Person -> JSObject
3  convertPersonToObject x = jsPerson ((toJSString .
       name) x) (age x)
4
5  foreign import javascript "{name: $1, age: $2}"
       jsPerson :: JSString -> Int -> JSObject
```

Listing 14. Receiving a `Person` record and marshal it into a JavaScript object

Marshalling JavaScript objects is not restricted to Haskell records, but it seems that it is one of the best fitting Haskell types. In section 8 there is an example where a JavaScript object is marshalled into a Haskell tuple.

JSFunction: As in Haskell, JavaScript functions are values and therefore can be passed as parameters or assigned to variables. Listing 15 shows how a `JSFunction` is taken as Parameter and executed as callback of the `setTimeout`[12] function.

```
foreign export javascript timeout :: JSFunction ->
    Int -> IO ()
timeout = setTimeout

foreign export javascript timeout5s :: JSFunction ->
     IO()
timeout5s = flip timeout 5000

foreign import javascript "setTimeout($1, $2)"
    setTimeout :: JSFunction -> Int -> IO()
```

Listing 15. Use a given JSFunction as callback of `setTimeout`

As described in this section, with type marshalling, it is possible to pass parameters and return types across the language barrier. This allows to cover a lot of potential use cases when working with Haskell and WebAssembly. But it requires some knowledge of JavaScript. Depending on the type, a lot of back and forth between Haskell and JavaScript is necessary. For larger data sets like large arrays of strings or nested JavaScript objects, this can be costly[I/O22].

To help developers which are not that familiar with the concept of type marshalling, Asterius could provide some more functions and explanations on how to deal with marshalling types, like it does for Strings. For example, with accessing attributes on a JavaScript object. Listing 16 provides two rather simple access function for object attributes. Note that the way of accessing the attributes have changed compared to Listing 13. This is because the notation in JavaScript depends on how this attribute is accessed. If the object's attribute is accessed directly from code, the "Dot-Notation" (`objectName.attributeName`) can be used. If it is accessed by a `String` variable, the "Bracket-Notation" (`objectName[attributeName]`) is necessary. The "Bracket-Notation" has to be used as well if the attribute name contains special characters, like spaces or hyphens.[Mor17].

```
-- accessing a property of type Int on a JavaScript
    object
accessObjectAttributeInt :: JSObject -> String ->
    Int
accessObjectAttributeInt obj key =  accessInt obj (
    toJSString key)

foreign import javascript "$1[$2] || 0" accessInt ::
     JSObject -> JSString -> Int


-- accessing a property of type String on a
    JavaScript object
accessObjectAttributeString :: JSObject -> String ->
     String
accessObjectAttributeString obj key = (fromJSString
    . accessString) obj (toJSString key)

foreign import javascript "$1[$2] || \'\'"
    accessString :: JSObject -> JSString ->
    JSString


-- accessing a property of type Double on a
    JavaScript object
accessObjectAttributeDouble :: JSObject -> String ->
     Double
accessObjectAttributeDouble obj key = accessDouble
    obj (toJSString key)

foreign import javascript "$1[$2] || 0.0"
    accessDouble :: JSObject -> JSString -> Double
```

Listing 16. Possible functions to give developers more convenience, when accessing attributes on `JSObjects`

8 Color - An extended example

In the previous sections the article described the bits and pieces, on how Haskell code is compiled to WebAssembly, how the resulting code is run in the browser, how single Haskell functions can be made available to JavaScript and how to deal with the type marshalling. Now it is possible to put it together in a more detailed example.

One of the possibilities to define colors in the browser is by a hexadecimal string representation of an RGB (Red Green Blue) color code. The first two digits of this code represent green, the second two represent red and the third two represent red. So the code `"FFFFFF"` would represent the color white and the code `"000000"` represents the color black. It is not uncommon for Web applications to calculate fitting colors based on an input color. For example, if a text has to be still displayed readable on an image which can be either of dark or of light color.

The following example provides the functionality to calculate the complementary color[13] to a given color and a function to calculate the appropriate brightness, black or white, for the text color. All based on a given color, either in hexadecimal or as an JavaScript object. The example uses Haskell language features which are not available in JavaScript like lazy evaluation, tuples and pattern matching. The resulting calculation functions will be made available in JavaScript as single functions.

Note: The code in this example does not claim to be the optimal way to solve the problem, but to demonstrate certain language features.

As a first part, in Listing 17 the initial definitions are done which are later used in the code. This part uses the Haskell features lazy evaluation and `type` declaration.

The first step is the module definition, which is necessary as described in section 6.

The second step is to define the necessary imports. In this case, `Asterius.Types` and Data.List. Those are used later on, to marshal the types. The imports follow two type aliases, with data for a Color-Tuple and the `JSHexcolor`.

[12] https://www.w3schools.com/jsref/met_win_settimeout.asp

[13] https://www.canva.com/colors/color-wheel/

The third step is to define of a list of tuples and its look up functions. The list of tuples is mapping a number (`Int`) to its hexadecimal representation (`Char`).

```
module Color where

import Asterius.Types
import Data.List

type Color = (Int, Int, Int)
type JSHexColor = JSString

-- Color is always a 6 digit Hex number:
-- first two stands for red,
-- second two for green,
-- third one for blue
hexMap :: [(Int, Char)]
hexMap = zip [0 ..] "0123456789ABCDEF";

hexToInt :: Char -> Int
hexToInt x = (fst . head . filter (\xs -> snd xs ==
    x)) hexMap

intToHex :: Int -> Char
intToHex x = (snd . head . filter (\xs -> fst xs ==
    x)) hexMap
```

Listing 17. The initial definitions of the color example.

In the Listing 17 the Haskell features type declaration, lazy evaluation are used.

The second part in Listing 18 defines some function to convert a color to its string representation and vice versa. To convert the code, it uses the language feature *"Pattern Matching"*.

```
-- Hex to Color conversion, and vice-versa
hexToColor :: String -> Color
hexToColor [] = error "Not a color string"
hexToColor [r1, r2, b1, b2, g1, g2] =
    (hexToInt r1 * 16 + hexToInt r2, hexToInt b1 *
        16 + hexToInt b2, hexToInt g1 * 16 +
        hexToInt g2)
hexToColor _ = error "Not a color string"

calcHexDigit :: Int -> String
calcHexDigit x = [intToHex (x `div` 16), intToHex(x
    `mod` 16)]

colorToHex :: Color -> String
colorToHex (r, g, b) = concat [calcHexDigit r,
    calcHexDigit g, calcHexDigit b]
```

Listing 18. Conversion of a color to its hexadecimal notation and vice versa.

In the third part, shown in Listing 19, the main logic to calculate the complementary color and the text color are defined. For the functions to be available in JavaScript the functions need to be exported, and its parameters have to be marshalled. To convert the type `JSString` to the type string, we use `toJSString` and `fromJSString` from the Asterius.Types module.

```
-- complementary color is on the otherside of the
    color wheel
complementaryColor :: Color -> Color
complementaryColor (r, g, b) = (255 - r, 255 - g,
    255 - b)

hexComplementary :: String -> String
hexComplementary = colorToHex . complementaryColor .
    hexToColor

-- if a background color surpasses a certain
    threshold,
-- either black or white text is better for
    readability
brightnessColor :: Color -> Color
brightnessColor (r, g, b) = color
    where
        threshold = (r * 299 + g * 587 + b * 114) `
            div` 1000
        color | threshold < 110 = (255, 255, 255)
              | otherwise = (0, 0, 0)

hexBrightness :: String -> String
hexBrightness = colorToHex . brightnessColor .
    hexToColor

-- populating the functions to be available in
    JavaScript
foreign export javascript jsComplementColor ::
    JSHexColor -> JSString
jsComplementColor = toJSString . hexComplementary .
    fromJSString

foreign export javascript jsBrightnessColor ::
    JSHexColor -> JSString
jsBrightnessColor = toJSString . hexBrightness .
    fromJSString
```

Listing 19. Calculation of the complementary color and the calculation of the text color.

So far, the example would be able to solve the given task based on a hexadecimal color representation. As an extension, it would be nice to have the ability to do the same thing, if we have the color in an object notation. For example: `{red: 255, green: 255, blue: 255}`. In this case, the Haskell code needs to receive a `JSObject` and convert it to a `Color` type in Haskell, then do the calculation and return a `JSHexcolor`. This is done in listing 20.

```
objectToColor :: JSObject -> Color
objectToColor input = (red, green, blue)
    where
        red = extractRed input
        green = extractGreen input
        blue = extractBlue input

foreign import javascript "$1.red || null"
    extractRed :: JSObject -> Int
foreign import javascript "$1.green || null"
    extractGreen :: JSObject -> Int
foreign import javascript "$1.blue || null"
    extractBlue :: JSObject -> Int

-- takeing a JSObject as an Input and return a color
    in Hex Form
foreign export javascript convertObjectToHex ::
    JSObject -> JSHexColor
convertObjectToHex = toJSString . colorToHex .
    objectToColor
```

Listing 20. The extension to receive the color in JavaScript object notation

As a second extension, a function should return the whole set of calculated colors, based on its input, as a JavaScript object. The given color itself, the appropriate text color, its complementary color and the appropriate text color of the complementary color. This is done in Listing 21, by calculating all necessary information and marshal the result to a JavaScript object.

```
1  -- put above functions together into an object
2  foreign export javascript jsColorInformation ::
       JSHexColor -> JSObject
3  jsColorInformation input = jsColorObject input
       brightness complementJS complementText
4      where
5          brightness = (toJSString . hexBrightness .
               fromJSString) input
6          complement = (hexComplementary .
               fromJSString) input
7          complementText = (toJSString . hexBrightness
               ) complement
8          complementJS = toJSString complement
9
10 foreign import javascript "{color: $1, textColor: $2
       , complementColor: $3, complementTextColor: $4}
       "
11     jsColorObject :: JSHexColor -> JSHexColor ->
           JSHexColor -> JSHexColor -> JSObject
```

Listing 21. Calculating all information and return them as a JavaScript object.

As a last extension, instead of receiving a single color, the program should have two functions that receive an array of colors in object notation. One function should return an array of the according hexadecimal notation of the colors, the other the color information as an object. Listing 22 shows the necessary functions to do the calculations and the according marshalling functions. Note that both functions have the same definition (`:: JSArray -> JSArray`). It might be advisable to define aliases for these types in Haskell for better readability: For example, `type JSHexArray = JSArray`.

```
1  foreign export javascript convertObjectsToHex ::
       JSArray -> JSArray
2  convertObjectsToHex input = value
3      where
4          marshalledInput = objectsFromJSArray input
5          objects = map objectToColor marshalledInput
6          colors = map (toJSString . colorToHex)
               objects
7          value = stringsToJSArray colors
8
9  foreign export javascript jsColorInformations ::
       JSArray -> JSArray
10 jsColorInformations input = value
11     where
12         marshalledInput = objectsFromJSArray input
13         objects = map (jsColorInformation .
               convertObjectToHex) marshalledInput
14         value = objectsToJSArray objects
15
16 -- Marshalling code copied and adapted from
       AsteriusPrim.hs
17 stringsToJSArray :: [JSString] -> JSArray
18 stringsToJSArray = foldl' js_concat_array_string
       js_array_empty
19
20 objectsToJSArray :: [JSObject] -> JSArray
21 objectsToJSArray = foldl' js_concat_array_object
       js_array_empty
22
23 objectsFromJSArray :: JSArray -> [JSObject]
24 objectsFromJSArray arr = [js_index_by_int arr i | i
       <- [0 .. js_length arr - 1]]
25 foreign import javascript "$1.concat($2)"
       js_concat_array_string :: JSArray -> JSString
       -> JSArray
26 foreign import javascript "$1.concat($2)"
       js_concat_array_object :: JSArray -> JSObject
       -> JSArray
27 foreign import javascript "[]" js_array_empty ::
       JSArray
28
29 foreign import javascript "$1[$2]" js_index_by_int
       :: JSArray -> Int -> JSObject
30 foreign import javascript "$1.length" js_length ::
       JSArray -> Int
```

Listing 22. Calculating all information based on a set of colors, both in hexadecimal and object notation.

Finally, the Listing 23 shows how the Haskell functions are called from JavaScript.

```
1  import * as rts from "./rts.mjs";
2  import module from "./color.wasm.mjs";
3  import req from "./color.req.mjs";
4
5  module.then(m => rts.newAsteriusInstance(Object.
       assign(req, { module: m }))).then(i => {
6      let randomColor = Math.floor(Math.random() *
           16777215).toString(16).toUpperCase().
           padStart(0, 6);
7
8      i.exports.jsComplementColor(randomColor)
9          .then(complement => console.log(complement))
               ;
10
11     i.exports.jsBrightnessColor(randomColor)
12         .then(textcolor => console.log(textcolor));
13
14     i.exports.convertObjectToHex({ red: 255, green:
           0, blue: 125 })
15         .then(hex => console.log(hex));
16
17     i.exports.convertObjectsToHex(
18         [
19             { red: 255, green: 0, blue: 255 },
20             { red: 255, green: 255, blue: 255 }
21         ]
22     ).then(hexes => console.log(hexes));
23
24     i.exports.jsColorInformations(
25         [
26             { red: 255, green: 0, blue: 255 },
27             { red: 255, green: 255, blue: 255 },
28             { red: 125, green: 255, blue: 125 }
29         ]
30     ).then(colorInformations => console.log(
           colorInformations));
31 });
```

Listing 23. Applying the compiled Haskell functions in JavaScript.

This example demonstrates, that it is possible to use Haskell language features like laziness, and strict typing to write a program, that interacts with JavaScript. It shows, that when

it comes to marshalling more complex types like arrays, a lot of additional code is necessary. Caution regarding type safety is necessary as well, as the values of the JavaScript types vary.

This example is available in Git alongside with the other code examples used in this article: https://bitbucket.org/nkarrer/haskell-and-webassembly

9 Conclusion

As this article has shown, starting to work with Asterius can be accomplished by people having some basic knowledge of Haskell, JavaScript, and HTML. First example applications can be compiled and used fast, thanks to the provided Docker image by Asterius and the output of the compiler. The documentation and tutorials provided Tweag I/O are informative and well written, but some information are outdated as details have changed during the implementation. For example, the tutorials mention the type `JSRef` which is now called `JSVal`.

As type marshalling is an important topic when working in the interface between the two languages Haskell and JavaScript, some further knowledge about the JavaScript Function Interface, its mechanics and its pitfalls, is necessary. Asterius could be more supportive in this field, by extending the documentation and providing additional functions. This article tried to close some of those gaps.

Even though the generated code HTML and JavaScript code is helpful at the beginning, it becomes noisy upon further usage. It would be good, if this code could be generated optionally.

With Asterius, the potential is given, that in the future there will be Haskell applications running in the browser and the NodeJS environment.

References

[con20a] MDN contributors. String.fromcodepoint(), 2020. last accessed 27.05.2022.

[con20b] MDN contributors. String.prototype.codepointat(), 2020. last accessed 27.05.2022.

[con20c] MDN contributors. String.prototype.concat(), 2020. last accessed 27.05.2022.

[Dev22] DevTut. Record syntax, 2022. last accessed 27.05.2022.

[Goo17] Google. Saying goodbye to flash in chrome, 2017. last accessed 09.05.2022.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

[I/O22] Tweag I/O. Javascript ffi, 2022. last accessed 27.05.2022.

[Mor17] Brandon Morelli. Javascript quickie— dot notation vs. bracket notation, 2017. last accessed 27.05.2022.

[Ora] Oracle. Java and google chrome browser, ? last accessed 09.05.2022.

[Sha18a] Cheng Shao. Asterius ghc webassembly backend reaches todomvc, 2018. last accessed 15.05.2022.

[Sha18b] Cheng Shao. Haskell webassembly calling javascript and back again, 2018. last accessed 15.05.2022.

[SHI19] Cheng Shao Sylvain Henry (IOHK). Haskell art in your browser with asterius, 2019. last accessed 15.05.2022.