CHALMERS | UNIVERSITY OF GOTHENBURG

Towards a Declarative Web

Master of Science thesis

Anton Ekblad

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, August 2012

Towards a Declarative Web

Anton Ekblad

Examiner: Koen Lindström Claessen

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2012

Abstract

Owing to its platform independence, ubiquity and ease of deployment, the web browser is quickly becoming not only a platform for large-scale application development, but *the* platform for application development. However, the de facto standard language of the web, Javascript, suffers from poor language design, a lack of static checks and a highly verbose programming model. Meanwhile, the Haskell functional language is gaining prominence as a language well suited for writing robust applications.

This thesis explores the viability of developing applications for the browser platform using Haskell. In doing so, it presents an implementation of a compiler from Haskell, including bleeding edge extensions specific to the state of the art GHC compiler, to comparatively lean Javascript code, together with a base library for writing web applications, based on Functional Reactive Programming.

Acknowledgements

I would like to thank Koen Claessen, for agreeing to act as my examiner, as well as for his help with administrative details and other aspects of getting this thesis done; Nick Smallbone for his supervision and valuable advice; Norman Ramsey for his great feedback during the presentation of this thesis; and Emil Vikström for proof reading and commenting on the early drafts.

Contents

1 Introduction

The modern computing landscape depends heavily on the web browser. We read the news and communicate with our friends in a browser; we watch TV and videos of adorable kittens doing the darnedest things in a browser; we file our taxes, collaborate on spreadsheets, listen to music, do our shopping and organize pictures of our children in a browser. Soon, we may even be voting in a browser! It is not unreasonable to say that, not only is the browser becoming an application platform, but it's becoming *the* application platform.

However, the browser is ill equipped to take on this role. The lingua franca of the browser, Javascript, was designed to be a lightweight way for non-programmers to add interactivity to the static web pages of the mid-1990's, and remains much the same, in spite of the applications being developed for the browser platform becoming ever more complex. Today, Javascript is being applied far outside of the domain it was designed for.[Severance; 2012]

Meanwhile, in the world of application development, functional programming is a rising star, claiming great improvements in stability, maintainability, defect rates and development time.[Hughes; 1984] Why not put this to work in the web application domain, where the need seems to be the greatest?

1.1 The Javascript problem

While it is possible to execute programs written in languages other than Javascript, such as Java, Flash or C#, in the browser, they all depend on the user having large third party runtimes installed and often suffer from compatibility problems. This effectively makes Javascript the only real option for writing programs for the browser platform.

That Javascript has issues is generally well known and acknowledged. Which of its properties are issues, and how bad these issues are, however, is under constant debate. It is clear, however, that there is ample room for improvements in the browser platform space. The properties of Javascript considered most problematic for the purpose of this thesis will be explained in detail in section 2.1.

1.2 Why Haskell in the browser?

While Javascript may be problematic as a language for developing large scale applications, it is not yet clear what should be done about it. There are many different languages that can be compiled to Javascript; why should Haskell be one of them?

- Functional programming has brought great improvements to many other areas, such as hardware design and verification, server-side web development, high assurance computing, etc. Considering that the browser is quickly becoming the major platform for application delivery, it makes sense to attempt to expand the reach of the functional paradigm there as well. Haskell is the de facto standard functional language for research and experimentation, making it a natural choice.

- Many of Javascript's problems stem from its poor type system and verbose syntax. Haskell, having an exceptionally expressive type system and fairly readable syntax, would seem like a good candidate to replace it.

- The GHC compiler offers fairly good bindings for use as a library, and its intermediate Core code is compact yet expressive. Building on the decades of work that have gone into it, it is possible to support all of Haskell2010, plus cutting-edge GHC-specific extensions, without having to discard and redo all of that work.

- As any higher level language running on top of Javascript will in all likelihood be slower than native Javascript, ensuring that such a language will not be too slow to be useful is important. Producing a highly optimizing compiler takes a substantial amount of work, and is outside the scope of this thesis, making leveraging GHC, which is known for producing very fast code, even more appealing.

- Web development is already one of Haskell's main areas of use.[Tibell; 2011] Thus, many developers would potentially benefit from a Haskell-enabled browser platform.

As we can see, there are many compelling reasons to attempt to apply Haskell in search of a solution to the complexities of modern web development.

1.3 Why yet another compiler?

However, even though we have established that it may be a good idea to enable the compilation of Haskell programs for the browser platform, it is still not clear that there is an actual need for another compiler; after all, there already exists numerous attempts at compiling Haskell to Javascript, as described in section 3 - are they not adequate?

Unfortunately, no. During the planning stage of this thesis, no complete compiler from Haskell to Javascript existed which was capable of producing output smaller than several megabytes for even the simplest example program, making them clearly unsuitable for anything but proofs of concept. At the time of writing, independent work on the GHCJS compiler has taken some steps towards remedying the situation, yet the basic problem of a major explosion of code, as well as serious performance and usability problems, still remain, as described in sections 3.1 and 4.3.

1.4 A solution made in Haste

This thesis explores the viability and pitfalls of using GHC as a vehicle for compiling Haskell to practically useful Javascript. It describes an implementation of a Haskell to Javascript compiler, dubbed Haste,[1] and a standard library, dubbed Fursuit,[2] for dealing with events and communicating with the user based on functional reactive programming.

As the goal of the thesis is to enable Haskell to be used for writing practical web applications, a set of goals can be derived: it should be possible to use the compiler and library to create arbitrary web applications; the produced code should not be prohibitively slow or large; and the compiler and library should be easy to install and use, not requiring any manual intervention to produce a

[1]HaSkell To Ecmascript compiler
[2]FUnctional Reactivity with Signals Using a sIngle Thread of control

working Javascript program, or manual patching of sources during the install process. More formally:

- a program compiled with Haste should not be slower than a functionally equivalent native Javascript counterpart by more than a constant factor of 10;
- nor should it be larger than an equivalent Javascript program by more than a constant factor of 10;
- compilation should produce a single self-contained Javascript source file, able to be executed by simply including it in an HTML document, without need for additional dependencies to be included;
- as the focus lies on producing a practically useful compiler within a reasonable time frame, more elaborate performance tuning, such as heavily optimizing or including profiling information in the produced Javascript code, is outside the scope of this thesis;
- similarly, compatibility with legacy web browsers, such as Netscape or Internet Explorer 6, is also outside the scope of this thesis.

2 Background

2.1 Javascript's shortcomings

As the purpose of this thesis is to improve many of the problems plaguing web development, understanding exactly what these shortcomings are, and why they should be remedied, is important. This section describes the rationale for why Javascript can be considered lacking as a language for large scale application development.

2.1.1 Bad scoping semantics

Javascript variables all live in the global scope, unless explicitly declared local. As declaring a variable is optional, unintentionally making supposedly local variables global, with hard-to-find defects as a result, is a real danger. Furthermore, unlike most other languages from the C family, curly braces do not open a new scope; the only local scope available is at the function level. Finally, whereas function calls are by value, variables captured by a closure get reference semantics. While this is not a defect per se, having the option of capturing certain variables by value would have been quite welcome.

2.1.2 Weak typing

Javascript employs dynamic, weak typing. While a static type system is often a great help in finding errors and reasoning about code, it's not really an option for an interpreted language like Javascript. The real problem, however, is brought on by the weak part of the type system; if an operation is applied to values with mismatching types, rather than throwing an error the runtime gleefully attempts to convert one or more of the values into something which is compatible with the other. This has interesting consequences, such as the equality operator not being transitive,[3] or the expression `(function(x) {return x+7;}) > [1,2,3]` always returning `true`, apparently indicating that functions are somehow greater than arrays.[4]

Combined with the fact that the addition operators + and += are overloaded for string concatenation, this makes for another class of subtle bugs. Consider, for instance, the following code snippet:

```
var age = prompt(“How old are you?”);
alert(“Really? Then you will be “+(age+1)+“ next year!”);
```

Assuming that the user gives her age as 45, this piece of code will happily inform her of her pending 451st birthday!

In addition, Javascript does not have the concept of integers. Instead, IEEE-754 double precision floating point numbers are used for all numeric operations, including array indexing. It goes without saying that this may sometimes be quite inconvenient.

[3]In Javascript, `(0 == "0" && 0 == "" && "0" != "")` is guaranteed to hold.

[4]For the curious, this is due to the comparison operators converting any incomparable operands to strings, then performing a lexicographical comparison on the two.

2.1.3 Poor support for the functional paradigm

Since Javascript has support for closures and first-class functions, it is often regarded as a functional language. However, its support for the functional paradigm is exceptionally poor. The specification makes tail call elimination impossible by requiring that the entire call chain is traversable, with function pointers and arguments of each call in the stack accessible; this means that continuation passing and efficient recursion is in general impossible.

The syntax for creating functions is also unnecessarily verbose; consider the following definition:

```
// Computes the sum of all even numbers in the given list
function sumEvens(xs) {
    var evens = filter(function(x) {return x%2 == 0;},xs);
    return foldl(function(acc,x) {return acc+x;},0,evens);
}
```

This function definition is quite cumbersome, even assuming that common functional primitives such as `filter` and `foldl` are already defined; unfortunately, they aren't in any major implementation. This lack of basic building blocks compounds the problematic syntax, as the programmer is forced to use it to define nearly every function she wants to use herself. While there are libraries that aim to remedy some of this, [jQuery] being the most widely used, there is no standard, and the ones that exist tend to suffer from *swiss army knife semantics*, that is, each function performs a wide array of operations, depending on their arguments. This makes code harder to read, write and maintain, as the actual effects of such operations depend on more factors and are thus harder to deduce. A prime example of this is jQuery's `map` function, which simultaneously performs the operations of Haskell's `map`, `filter` and `concat` functions, depending on the arguments given.

Finally, Javascript also offers no support for enforcing immutability, discouraging programming techniques such as single static assignment and general coding best-practices.

2.1.4 Lacking modularity

Unlike nearly every other programming language since C, Javascript does not support modules or include files, making it extremely cumbersome to link individual pieces of code together. While there are partial workarounds, such as the module design pattern [Cherry; 2010], the developer is still forced to litter her HTML documents with `<script>` tags for every piece of code she wants to include. Not only does this create unnecessarily tight coupling between HTML and Javascript, but is also fragile and prone to human error.

2.2 STG as a source language

Haste uses GHC's STG intermediate language, as described in [Peyton Jones; 1992], as the starting point for its code generation. While not as widely known as the *Core* or *Cmm* intermediate languages, STG has benefits over both for the purpose of generating code in a comparatively high level target language.

STG is very similar to Core; in fact, the code generator was initially written to use Core as its input language, and later on ported to STG in a matter of hours. It has one compelling benefit though: in STG, thunks and evaluation are explicit, whereas in Core, they are not. Thus, using STG better leverages the hard work GHC puts into strictness analysis and optimization. Another consequence of this is that STG quite naturally supports unboxed values. In addition, STG expressions also include some lower level information Core does not, such as the list of variables captured by a closure.

Cmm, in contrast, is a GHC specific dialect of C–, a low level assembly language with syntax similar to C. In spite of being a good intermediate target when the final target is native machine code, using Cmm as a starting point would lose higher level information about code structures such as closures and case expressions, forcing the code generator to implement them using lower level code than what would be necessary with Core or STG, making the resulting code larger and slower in the process. Another major deal breaker is Cmm's use of labels and unstructured jumps, something which Javascript lacks entirely.

The STG language contains the following constructs:

- *Let-binding*; these represent bindings of data constructors, closures and thunks[5] to variables in the context of an expression, thunks being a special case of closure. Apart from the arguments and body of the closure, closures also contain the update flag, which tells whether the closure represents a thunk or not, and a list of any free variables in the closure's body. Let-bindings may be recursive or non-recursive, specifying an entire group of one or more mutually recursive bindings in the recursive case.

- *Function application*; the application of a function to one or more arguments. Unlike Core, STG allows functions to have any number of arguments, including zero. Applications of data constructors and primitive operations must be fully saturated, whereas partial application is allowed in the general case. Only literals and named variables are allowed as function arguments, forcing more complex arguments to be bound to a variable before use.

- *Case expression*; compares an arbitrary STG expression to a series of patterns with corresponding branches, executing the branch belonging to the first matching pattern. Case expressions must be complete, in that there must always be a matching branch for the expression being scrutinized.

- *Atoms*; variables and literals, plain and simple.

[Peyton Jones; 1992] conveniently describes the semantics for each of these constructs, making the STG language a relatively well documented and precisely defined intermediate language, further affirming its suitability for the purpose of code generation.

[5]The basic construct used to implement lazy evaulation. It essentially contains a pointer to a nullary function which, when the thunk is evaluated, gets called and replaced by a pointer to its return value.

2.3 Functional Reactive Programming

Functional Reactive Programming, or FRP for short, as formulated by [Elliott, Hudak; 1997], centers around the concepts of *behaviours* and *events*. Events are best viewed as a stream, or lazy list, of timestamped values of the same type, each value denoting one occurrence of said event. Behaviours can be thought of basically as functions over time. Behaviours are sampled using the `at :: Behav a -> Time -> a` function at given points in time, yielding a construct the value of which varies depending on the time of sampling and the occurrence of events prior to that point in time. Individual occurrences of events are sampled using the function `occ :: Event a -> (Time, a)`. The key concept for this formulation of FRP is that behaviours essentially react to events in a manner dependent on the time at which the behaviour is sampled and the history of event occurrences.

A better understanding may be gained by examining some of the FRP primitives presented in [Elliott, Hudak; 1997].

The most basic behaviour is `time`, which simply returns the time at which it was sampled. Its semantics are more formally given by:

```
time: Beh Time
at [time] t = t
```

Further, a way is needed to apply functions over static values to arbitrary behaviours; something which is accomplished by `lift`$_n$. It is interesting to note that the interface provided by the `lift` functions corresponds closely to that implemented by `Functor` and `Applicative` in Haskell, an application `lift3 f a b c` being equivalent to `f <$> a <*> b <*> c`. In fact, in later formulations of FRP, applicative functors are used for exactly this purpose.

```
lift_n:
  (a_1 -> ... -> a_n -> b) -> Beh a_1 -> ... -> Beh a_n -> Beh b
at [lift (f a_1 ... a_n)] t = f (at a_1 t) ... (at a_n t)
```

The key combinator for this formulation of FRP is the `untilB` function, which enables behaviours to change dynamically depending on the occurrence of events. It takes an initial behaviour b and an event e which delivers new behaviours on every occurrence. Its semantics are given by:

```
untilB: Beh a -> Evt (Beh a) -> Beh a
at [untilB b e] t = if t <= tₑ then at [b] t else at [b’] t
  where (tₑ, b’) = occ [e]
```

However, this combinator is sometimes problematic, in that it causes *time leaks*, a phenomenon where a behaviour needs to traverse, and consequently hang on to for the duration of the program, the entire stream of events in order to determine how to act when sampled.[Apfelmus; May 2011]

Behaviours being continuous functions over time has historically complicated efficient implementations of FRP; unlike other reactive programming models, such as GUI libraries, behaviours are continuous over time, meaning that the simple solution of idling until a new event occurs and then computing its results is not applicable. Thus, older FRP implementations tended to be pull-driven, sampling behaviours at regular intervals, recomputing inputs that may not even have changed in the process, which is obviously rather wasteful of system resources.

[Elliott; 2009] present an efficient formulation of FRP by separating out continuous time functions from purely discrete reactive behaviours, using sampling for the former while employing an efficient, push-based implementation for the latter. The paper also uses type class morphisms to give the FRP combinators a facelift, using the now well established interfaces for functors, applicative functors and monoids, introducing a more accessible and familiar interface.

3 Related work

Lots of work has been done both in the field of Haskell to Javascript translation and in the FRP space. While it was ultimately decided, for reasons described in the following subsections, to not build neither the compiler nor the reactive library on top of any of these existing solutions, their implementations still make use of many of the ideas explored by these previous works.

3.1 Haskell to Javascript compilers

While there have been several attempts at translating Haskell into Javascript, the first one being [Golubovsky; 2007], at the time of writing, there exists no completely satisfactory solution. The large amounts of such projects, however, provides an abundance of ideas and examples to incorporate, and ample information on what pitfalls and design decisions to avoid.

The rest of this section examines the three most prolific such projects, GHCJS, UHC and YHC, as well as some of the less visible contributions in lesser detail.

3.1.1 GHCJS

GHCJS [Nazarov; et al], is fairly similar to Haste, and indeed served as inspiration for some of its design decisions; it uses GHC as a front end, and Javascript code is generated from the resulting STG. During the planning phase of this thesis, GHCJS was seemingly unmaintained and fragmented into several mutually incompatible forks, all producing highly problematic output, being several megabytes in size and requiring a lot of time consuming and poorly documented manual assembly in order to actually execute. This, combined with certain technical decisions that were deemed problematic for the purpose of this thesis, and will be explained shortly, made GHCJS an undesirable starting point for a practical compiler.

Recently, the project has refocused and addressed some of these shortcomings. In particular, it supports concurrency, weak references with finalizers, on-demand loading of code over a network connection and post processing using the Google Closure compiler. It also comes with a custom *cabal* patched to install intermediate code for libraries alongside their native counterparts, allowing for very user friendly package management. This would indeed seem like a good starting point for a practical Haskell to Javascript compiler. However, there are several reasons why was not chosen for this thesis.

- While not as drastic as before, code footprint is still huge; on the order of several hundred kilobytes for a trivial example program.

- The decision to support concurrency is problematic. It greatly complicates the runtime and calling conventions for functions in order to simulate threading on top of Javascript; not only does this added complexity increase the potential for subtle bugs, but it makes it harder to reduce code footprint as well. While I have not yet been able to get GHCJS to produce complete, executable code, in spite of considerable effort, an inspection of its output reveals that programs compiled with it are likely to be several times slower than when compiled Haste, due to having to eschew many optimizations, of function calls in particular, in order to support concurrency. While concurrency can sometimes be quite useful, particularly in server applications having to handle multiple connections, web applications rarely perform this kind of functionality, making the added complexity of perceived non-determinism[6] and overly complicated runtime too large a price to pay.

- Although on-demand loading of code is a useful feature, the way it splits the generated code into several independent bundles makes reduction of code footprint and pruning of unused code very hard. While the GHCJS developers have done much to alleviate this problem, its code footprint is still prohibitively large.

- GHCJS generates plain text Javascript immediately from STG, rather than taking a detour via a symbolic intermediate representation. This makes many optimizations and simplifications of the resulting code impossible to make.

- The build process and workflow of GHCJS are poorly, even incorrectly, documented. Compiling a working binary requires non-trivial knowledge of the GHC build process and a willingness to experiment, while its output code requires a fair amount of manual assembly to get in working order.

[6]As Javascript does not support parallel code execution, truly non-deterministic code is not possible. However, the complexity for the application programmer in dealing with multiple threads of execution remains.

Even though these properties make GHCJS unsuitable for the purposes of this thesis, there are many lessons to take away from GHCJS, particularly the decision to draw upon the GHC compiler as a library, the benefits of harnessing existing tools for Javascript augmentation and the integrated package management.

3.1.2 UHC

The *UHC* compiler, developed at Utrecht University, supports compilation to Javascript in recent versions.[Dijkstra; 2010] In addition to having first-class support in the compiler proper for Javascript generation, which is notable in itself, it also has a fairly extensive runtime library, supporting many of the technologies routinely used by web developers, and a lot of work and documentation put into getting its foreign function interface working nicely with Javascript.[Stutterheim; 2012]

Unfortunately, UHC generates *very* slow code even when compiling for a native target [Seipp; 2010], which shows in its generated Javascript as well, as we shall see in section 4.3. It also does not implement the entire Haskell2010 specification, and suffers from the fact that GHC is the de facto standard compiler for Haskell and that a lot of programs depend on GHC-specific extensions to the language specification, most of which are not supported by UHC. In addition, the Javascript UHC generates is very large, the simplest example programs resulting in nearly two megabytes of code, with some trivial, but tedious, manual assembly required, as the generated code attempts to pull in its dependencies from Javascript files on the local system, using their absolute, system-dependent paths.

3.1.3 YHC

The first, and arguably most practical so far, tool to enable Haskell code to run within the browser was the *YCR2JS* translator, which converted the binary Core intermediate output from the YHC compiler into Javascript.[Golubovsky; 2007] Like UHC, it sports a considerable library although with a focus on higher level widgets rather than web-specific browser primitives. It also generated smaller code than any of the other major compilers, tending towards an output code size of around 100 kilobytes for trivial example programs which, while still a bit on the large side, is not quite as dramatic as the multi-megabyte code dumps produced by UHC. While its produced code is a bit on the slow side, it is still not prohibitively so.

Unfortunately, just like UHC, YHC suffered from not being GHC, lacking extensions many programmers had come to rely on. The compiler itself was also never finished, and was discontinued shortly after its Javascript backend was introduced.[Mitchell; 2011]

3.1.4 Others

Apart from the "Big Three", there have been several other attempts at tackling the problem, including at least three bachelor's theses at the Computer Science and Engineering faculty at Chalmers and Gothenburg University. Two of which, *Haskell in Javascript* by [Bengtsson, Bung, Gustafsson, Jeppsson; 2010] and *JSHC* by [Björnesjö, Holm; 2011], are written entirely in Javascript, allowing the compilers themselves to run in the browser, and implement GHCi-like graphical user interfaces, allowing for real-time experimentation with Haskell within the browser. Neither supports standalone compilation though, nor implements more than a subset of the Haskell98 specification.[Björnesjö, Holm; 2011] The third, *LambdaScript* by [Ekblad; 2011], does support stand-alone compilation, but only supports a context-free Haskell-like syntax rather than proper Haskell, does not have the user-friendly graphical interfaces and ability to run within the web browser of the other two, and lacks important functionality such as type classes.

Other attacks on the problem include using an embedded domain-specific language to generate Javascript from Haskell [Björnsson, Broberg; 2008], and using the *Emscripten* LLVM to Javascript compiler [Zakai; 2011] to compile the output of GHC's LLVM code generator into Javascript. The former approach still keeps Javascript semantics, however, and the latter would require the compilation of the entire GHC run time system into Javascript, which would limit the usefulness of such a solution.

3.2 FRP

There are a multitude of different implementations of the general idea of functional reactive programming, most of which build upon the ideas presented in [Elliott; 2009]. The most well known ones are Elliott's own *reactive* library, and *reactive-banana* by Heinrich Apfelmus [Apfelmus; March 2011], which are briefly described in the following sections.

Unfortunately, as both build on GHC's concurrency primitives, neither is suitable for use in Haste's non-threaded environment without extensive patching.

3.2.1 reactive

Conal Elliott's *reactive* library is a fairly straightforward implementation of the concepts presented in [Elliott; 2009]. It contains a vast array of combinators for performing various operations on reactive values, and makes heavy use of type class morphisms to provide a familiar interface. Unfortunately, the project has not seen any updates in the last two years, and apparently contains some rather serious bugs that make it unsuitable for production use.

It contains a set of adapter functions, for use with imperative code, from which certain ideas can be gleaned, however. In particular, the notion of *sinks*, value consumers that basically attach an IO action to a reactive value, and the idea of using an imperative-style timed callback to provide a notion of time to otherwise discrete reactive values, are rather useful.

3.2.2 reactive-banana

In contrast to *reactive*, which is a rather strict interpretation FRP theory, *reactive-banana* takes a more pragmatic approach. It acknowledges the problem with time leaks mentioned in section 2.3, and used to eschew the `untilB` combinator, renamed `switcher` in more recent FRP literature for this very reason. More recent versions introduce a restricted version of it however, employing the same phantom types trick as the ST monad to prevent time leaks.[Apfelmus; 2012]

Its set of combinators is also rather different from the one provided by *reactive*. Rather than providing every function imaginable, *reactive-banana* provides a fairly basic set of combinators, closely matching the ones found in [Elliott; 2009], instead providing a greater range of tools for manipulating when, and how often, events occur as viewed by the library's behaviours.

3.3 Google Closure compiler

Many, if not all, of the services provided by Google make heavy use of Javascript. As these services are used by a vast number of people every day, even small increases in their code size can be expensive. Additionally, as we have seen, Javascript comes with quite a few pitfalls when used to develop non-trivial applications. In order to combat these problems, Google created the Closure compiler; a tool which parses, analyzes and type checks Javascript, warning the user of any potentially incorrect code it detects, and then outputs an optimized version of its input Javascript. The compiler, among other optimzations, shorten variable names, strip comments and unnecessary whitespace, inlines suitable functions and removes dead code.[Google Closure Compiler]

In order to leverage this work already done on the optimization of Javascript programs, Haste optionally makes use of the Closure compiler during its post-processing step.

4 Implementation

The implementation takes the form of two main components: a compiler, consisting mainly of a code generator and a skeleton for calling upon GHC to perform the rest of the work, and a push-driven reactive library, specifically tailored to run within in a non-threaded environment.

4.1 Code generator

The code generator is implemented as a single pass over the GHC's intermediate STG representation, translating the code into an intermediate symbolic

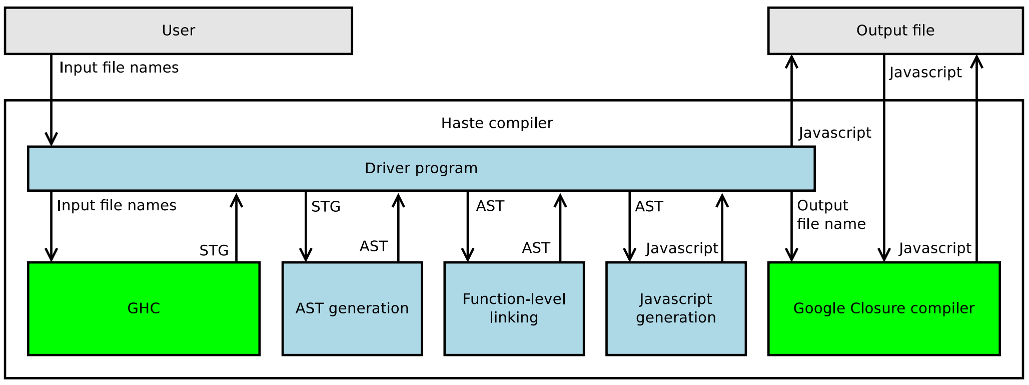

Figure 1: Information flow between components during compilation. Gray boxes represent external entities, green boxes represent third party components, and blue boxes represent internal Haste components.

representation of the Javascript code to be emitted at the end of the translation, hereafter referred to as the *AST*, on which certain simplifications are performed, to enhance performance and readability, and reduce code size. The basic blocks of this intermediate representation are written to a custom Writer monad as the STG structure is traversed. Then, a function-level linker knits together all code needed by the final program and the result is translated from the symbolic representation. Finally, the Google Closure compiler is (optionally) executed on the resulting Javascript, to further shrink and optimize the output. The information flow of the compilation process is visualized in figure 1.

The following sections describe the phases of translation mentioned above, the compiler's internal data representation, how interfacing with Javascript code is handled, and the runtime system driving the resulting code.

4.1.1 The AST

Haste uses a symbolic intermediate representation as a stepping stone from STG to Javascript, to allow for optimizations and simplifications that only makes sense in an imperative, or even Javascript-specific, context to be performed after the translation from STG has taken place. This intermediate representation comes in the form of two major algebraic data types, `JSStmt` and

`JSExp`, representing Javascript statements and expressions respectively, along with a few minor ones, for representing names, operators, etc. These data types are collectively known as “the AST”, an abbreviation of *Abstract Syntax Tree*. The AST contains operations that translate directly into a subset of Javascript, including `while`-loops, `continue`-statements,[7] `if`, conditional operator[8] and `switch-case` conditionals, `return` from function, variable and closure creation, most arithmetic and logic operators, and array creation and indexing. In addition, the AST also supports some higher level constructs.

4.1.1.1 Thunks

While thunk creation and evaluation could easily be implemented using generic closures and function calls, are common enough to warrant their own explicit `Thunk` and `Eval` operations. In particular since they require some special machinery to handle updates properly. While a normal closure has arguments and a function body, a thunk has a body and a *final expression*; the final expression is the value the thunk will be updated with after its body has executed, and will most commonly refer to a local variable.

4.1.1.2 Function calls

In Haste, function calls come in four flavours: normal calls, fast calls, native calls and native method calls.

A *normal* call is handled by the runtime’s eval-apply machinery, and is described in further detail in section 4.1.9.3.

A *fast* call is a function application that Haste knows for sure is neither over nor undersaturated. At run time, these are normal Javascript function calls, without any RTS machinery to get in the way.

A *native* call is a call to a non-Haskell function; rather than a variable or closure, the callee of a native call is a simple string, containing the name of the function to be called. Haste uses these for primitive operations and foreign imports. Native calls execute exactly like fast calls.

[7]In Javascript and many other C-like languages, the `continue` keyword transfers program control to the head of the currently executing loop, as a restricted form of semi-unstructured jump.

[8]Equivalent to Haskell’s `if-then-else` construct, the ternary operator of the C family takes the form of `cond ? thenExpr : elseExpr`.

A *native method* call is the same thing as a native call, except with an object. Thus, the AST code

```
NativeMethCall expr “method” [arg1, arg2]
```

will compile into

```
(expr).method(arg1, arg2)
```

4.1.1.3 By-value capturing closures In Javascript, variables captured by closures have reference semantics; update such a variable in one place, and every closure that has a reference to it sees the change. This is sometimes not what we want, however; Haskell having only immutable values, reference semantics sometimes cause problems with the generated code, in particular when optimizing tail recursion into loops. *Const closures* solve these problem by simulating by-value capture of a closure's free variables. They, along with the problem that prompted their inclusion, are described in further detail in section 5.2.

4.1.2 Intermediate code generation

The generation of AST from STG takes place within a specialized variant of a Writer monad; `JSGen`. Code executing within the monad may `emit` AST statements, which are then appended to the *code stream.* The code stream is a Bag, a data structure which allows constant time append and prepend operations, and represents the current expression being processed. The monad also keeps track of which variables have been accessed within the current function, as well as which have been created, to help construct the dependency graph at the end of code generation; the name and arguments of the function currently being generated, to ease optimization of tail recursion; and the code generator's configuration, received from the compiler driver program and created from the command line arguments passed by the user.

Code is generated for one top level definition at a time, even with groups of mutually recursive functions. Each top level definition is then paired up with the set of global variables its body touches, to enable dependency walking during link time, which will be described later on.

One point of friction between STG and Javascript are statements; in STG, everything is an expression, whereas in Javascript, statements are needed to implement many STG expressions. In general, code generator functions *emit* statements, but *return* expressions. The relationship between the emitted statements and the returned expressions can be thought of as implementation and interface respectively; the emitted statements perform some computation, and the returned expression represents the value resulting from that computation. Usually, a function that emits statements will emit a final statement that assigns the result of the computation performed to a variable, and then return that variable.

Emitting statements this way makes it hard to generate code recursively, though. For this purpose, the `JSGen` monad provide a primitive to generate statements without writing them to the code stream, instead returning them alongside the expression, which we shall examine further in the context of the following example.

```
foo x = case x of
          Just x' -> x'*x'
          Nothing -> -1
```

Clearly, the AST primitive we emit for this `case` expression must contain whatever code we generate for its branches. In this situation, the code generator uses the monad's `isolate` primitive. `isolate` takes a `JSGen` computation and executes it within an isolated environment, without it being able to neither see nor affect the environment of the calling computation, returning any generated code along with the computation's return value. The only thing that propagates from the `isolated` environment are the variables touched and created, as we need a complete set of every function's dependencies to construct the dependency graph at the very end. Thus, in pseudo code, generating code for the above example would look as follows.

```
push (foo, [x]) onto the function stack
body <- isolate $ do
  touchVariable x
```

```
    scrutinee <- isolate $ generateScrutinee x
    branch1 <- isolate $ generateCaseAlt (Just x' -> x'*x')
    branch2 <- isolate $ generateCaseAlt (Nothing -> -1)
    emit (Case scrutinee [branch1, branch2])
  discard (foo, [x]) from top of function stack
  foo' <- generateVariable foo
  emit (Assign foo' (Fun [x] body))
```

The above pseudocode demonstrates most of the important functions of JSGen; the function currently being analyzed, along with its arguments, is pushed onto a stack of functions, as described above. Then its body is generated in isolation, inside which subexpressions may be further isolated. When the code generator is done with the function, it pops the top of the function stack and assigns the function to a variable.

Now that we are somewhat familiar with the JSGen monad, we shall take a look at how the various expressions of the STG languages are processed. Each description is accompanied by pseudocode describing the transformation. The structure of the pseudocode is not the same as in the actual compiler, but simplified in order to make the description of each transformation clearer.

4.1.2.1 Variable bindings The variable being bound, as well as its arguments, if any, are pushed onto the function stack. Unless the binding is on the top level, it is marked as a locally created variable. The binding's right hand side is generated within this environment, after which the assignment of the right hand side to the variable being bound is emitted.

If the binding being generated is tail recursive, wrap the right hand side in a const closure.[9] The right hand side may be either a closure or a constructor application.

```
  genBind var rhs = do
    pushBind (var, argsOf var)
    when (not topLevel) (addLocal var)
```

[9]See section 5.2 for a detailed discussion of tail recursion and const closures.

```
    rhs' <- isolate $ genRHS rhs
    let rhs'' = if tailRecursive var
                  then ConstClosure (freeVars rhs) rhs'
                  else rhs'
    emit (Assign var rhs'')
    popBind
```

4.1.2.2 Closures and thunks Generate the abstraction body; if the abstraction happens to be a thunk, return a thunk containing the statements generated from the closure's body as the thunk's body, and the expression returned alongside it as the thunk's final expression.

```
genClosure args body = do
  (body', expr) <- isolate $ genExpr body
  if null args && isUpdatable body
    then return (Thunk body' expr)
    else return (Fun args (body' ++ [Return expr]))
```

4.1.2.3 Function applications If the caller and the callee are the same, and if we are in tail position, emit reassignments of all of the function's arguments followed by a `continue`. Otherwise, if the function's arity as given by its STG representation matches the number of arguments given, return a fast call, otherwise return a normal call. If general tail call elimination via trampolining is enabled, wrap the returned function call in a thunk.

```
genApp callee xs = do
  (caller, args) <- getTopOfFunStack
  if caller == callee
    then do
      emit (zipWith Assign args xs)
      emit Continue
    else do
      if useTrampoline
        then return (Thunk theCall)
```

```
          else return theCall
    where
      theCall
        | arityOf callee == length xs = FastCall callee xs
        | otherwise                   = Call callee xs
```

4.1.2.4 Case expressions For case expressions, first two new variables are created; the scrutinee, which will hold the value of the expression being examined, and the *result variable*, which will hold the resulting expression of the branch taken. Then, code for the expression to be scrutinized is emitted, adding explicit evaluation if trampolining is activated, followed by the code for the alternatives where each alternative is responsible for checking if they match the scrutinee and, bind any variables pattern matched against and binding their result to the result variable.

The generated alternatives are then examined for optimization opportunities; if there is only a single branch, the case expression is purely for evaluation or deconstruction by pattern matching, so only the bindings and body of the single branch are emitted; if there are two branches, an `if-then-else` structure is emitted; otherwise, a `switch-case` structure is emitted.

```
genCase expr alts = do
  scrut <- newVar
  resultVar <- newVar
  expr' <- genExpr
  alts' <- mapM (genAlt scrut resultVar) alts
  case alts of
    [alt] ->
      emit (bindings alt) >> emit (body alt)
    [(condIf, ifBranch), (_condElse, elBranch)] ->
      emit (If (compare ifAlt scrut) ifBranch elBranch)
    _ ->
      emit (Case scrut alts)
  return resultVar
```

4.1.2.5 Data constructors, PrimOps and others There are a few constructs in STG that, being quite simple in their nature, do not warrant their own detailed description. Literals are simply copied verbatim into the AST. Constructor applications are always saturated, meaning that a constructor application is just an instantiation of the data format described in section 4.1.7. If the constructor application is recursive, it is wrapped in a thunk, otherwise it is inlined in order to reduce code size. This inlining is safe, as STG guarantees that constructor arguments are always literals or variables, meaning that inlining constructor applications will never cause evaluation to take place.

Primitive operations are generated either as native Javascript operators or as native calls to functions; the `NativeCall` and `NativeMethCall` AST operations mentioned in section 4.1.1.2 exist mostly for this purpose, and their use is trivial, as STG guarantees that application of primitive operations is always fully saturated. This property is what enables the generation of many primitive operations as native Javascript operators as well.

As Javascript does not have an integer type, using its native Number type, corresponding to IEEE-754 double precision floating point numbers, primitive operations on fixed precision integers have some caveats:

- All arithmetic operations receive an extra `|0` operation. In Javascript, bitwise operations convert their operands to 32-bit signed integers, making bitwise OR:ing with 0 a compact way to ensure that arithmetic is always performed modulo 2^{32}. There are still issues with multiplications, however, which are very hard to solve in an efficient manner. These issues are described in more detail in section 5.4.
- Bitwise operations on unsigned values are somewhat problematic in Javascript, as binary operations are performed on 32-bit signed numbers only; this issue is discussed in greater detail in section 5.1.

Calls to foreign imports are similarly trivial, compiling into a simple `NativeCall` operation, as STG, again, guarantees that applications of foreign functions are always saturated.

4.1.2.6 Output The intermediate code generation stage outputs a `JSMod` structure for each module, containing the module's name, the list of AST functions defined in the module, and the set of variables each of the functions depends on. If `hastec` is invoked with the `--libinstall` flag, this is where the compilation process ends, and the `JSMod` structure is written to the module cache, for inclusion by other programs.

4.1.3 Tail call elimination

Haste comes with two flavours of tail call elimination. Tail recursion elimination, and full general call elimination using trampolining.

4.1.3.1 Tail recursion elimination As it does not negatively impact code size, performance or clarity - quite the opposite, in fact - tail recursion elimination, TRE for short, is always enabled, and can't be turned off. It transforms every eligible function into a loop, replacing the tail recursive call(s) with a reassignment of the function's arguments followed by a `continue` statement, transferring control flow to the top of the loop.

To be eligible for TRE, a function must call itself in tail position on at least one branch, and not be mutually recursive. The requirement on eligible functions to not be mutually recursive is not intrinsic to the problem, but prevents TRE from interacting badly with the data constructor inlining described at the end of section 4.1.2.5; having both enabled at the same time caused subtle bugs in the *integer-simple* package that didn't show up in any other, less complex code. As both optimizations were deemed valuable, disallowing TRE in the small subset of functions that are both tail recursive *and* mutually recursive was considered an acceptable loss. The implementation of TRE is further discussed in section 5.2.

4.1.3.2 General tail call elimination Disabled by default, users can enable full tail call elimination by passing the `--opt-tce` flag to the compiler. When enabled, all function calls are wrapped in thunks, and the runtime uses a special trampolining evaluation function that evaluates thunks in a loop for as long as they return a new thunk, only returning when a thunk give back a value rather than another thunk. This solution adds very little to the complexity of the code, as it leverages the existing lazy evaluation machinery for trampolining, the overhead it adds not only hurts performance, adding at least one extra function call and several comparisons to every call, but also seems to make things harder for the Google Closure compiler, which is used in the postprocessing stage to shrink the generated code.[10]

4.1.4 Linking

After intermediate code generation, unless invoked with the `--libinstall` flag, Haste links together all modules needed to produce an executable Javascript blob. This linking happens on the function level, basically entails walking the dependency tree of all involved modules. Using `Main.main` as the starting point, the linker looks at the dependency tree of each dependent symbol and includes them, recursively working their individual dependency trees as well. If a dependent module is not found in the source tree being compiled, Haste looks for it in the module cache, which usually resides in `~/.haste/lib`.

Walking the dependency tree this way, only including the functions encountered, guarantees that the final Javascript blob will not include any code that is not actually reachable from the program's entry point.

When all functions to be included in the final code have been gathered their AST representations are sent on to the pretty printer, the results of which is then written to an output file.

4.1.5 The Pretty Printer

The pretty printer is the part of Haste responsible for converting AST code into actual textual Javascript. It has three different modes; pseudo, pretty and compact. The *pseudo* mode, quite unsurprisingly, outputs pseudocode that is not executable. All names from the original STG code are preserved, and the code is indented and formatted for maximal readability. This mode is used by the

[10]Tests on the *Glosie* application presented in section 4.3.3 indicates that the post-Closure code output, for a code base which barely uses tail calls at all, grows by more than 12%, while comparing the runtimes of the *fibs* microbenchmark with and without trampolining reveals a stunning *1500% execution time penalty* for the trampolining version!

`hastecat` tool, which lets the programmer view the pretty-printed source code of precompiled AST modules.

The *pretty* mode produces actual runnable output, but tries to strike a balance between readability and code size. The code is indented and formatted using tab characters, and the names of all external STG symbols[11] appear next to their new Javascript names as comments, making it very useful for debugging the code generator, trying out changes in the RTS or just studying the code generator and its output. This mode is used when invoking `hastec` with the `--debug` flag.

The *compact* mode is the default mode used when invoking `hastec`. It does away completely with comments, indentation and formatting, focusing on shrinking the code as much as possible. While this is the most compact mode available, it is also completely impossible to read, and thus useless for debugging.

The output of the pretty printer is, as described in the previous section, written to a .js file. It is the final stage of compilation, unless postprocessing is enabled by invoking `hastec` with the `--opt-google-closure` flag.

4.1.6 Postprocessing

Even though Haste produces very small output compared to other similar compilers, as we shall see in section 4.3, it still generates far from perfect code. In particular, many redundant assignments are generated due to the fairly straightforward translation from the extremely `let`-happy STG source. Instead of spending considerable time and effort optimizing away these kinks, we can stand on the shoulders of a certain giant: Google.

When invoked with the `--opt-google-closure` option, Haste runs the Closure compiler, described in section 3.3, on its output as the final step of compilation. While it seems that the inlining capabilities of Closure do little to reduce the execution time of programs, GHC having already performed significant inlining, the effect on code size is dramatic: compiling the *Glosie* program, introduced in section 4.3.3, with `-O2 --opt-google-closure` reduces

[11]That is, the ones that have meaningful names, that may be linked to the original Haskell code in some useful way.

the output code size by more than half, compared to compiling with just -O2!

Additionally, the type checking and code analysis performed by Closure serves as a sanity check when making changes to the code generator; if it suddenly starts producing warnings where it did not before, putting extra effort into verifying the correctness of the changed code becomes even more important than usual.

4.1.7 Data representation

As Haskell supports more complex data types than just numbers and strings, more specifically algebraic data types at the programmer level, and thunks at the operational level, representing values of these types in Javascript becomes an issue. Additionally, as there is an impedance mismatch between some of the primitive types of Haskell and the types available in Javascript, thought needs to be given to their representation as well.

4.1.7.1 Primitive types Haste supports the five primitive types of Haskell: `Char`, `Float`, `Double`, `Bool` and `Int`, with `Int` being defined as having a precision of 32 bits. All boxed primitive types except `Bool` use the representation of algebraic types presented in section 4.1.7.2; an `Int` is, as usual, represented as `I# Int#` in Haskell syntax, a `Double` as `D# Double#` and so on. The representation of data constructors is described in detail in section 4.1.7.2. For the remainder of this section, all references to the primitive types other than `Bool` refer to their *unboxed* representation.

All of the unboxed numeric types are represented by Javascript's `Number` type, making `Float#` and `Double#` basically synonyms; coercing between the two is completely safe. For `Int#`, this representation does not guarantee that what is supposed to be an `Int#` actually is; this is instead guaranteed by all primitive operations on `Int`s, which ensures that operations are always performed modulo 2^{32}.

`Char#` is represented by a single character string, using Javascript's `String.charCodeAt` and `String.toCharCode` for conversions to and from `Int#` respectively.

`Bool` is treated slightly different from the other primitive types. In order to make comparisons more efficient and readable, both boxed and unboxed booleans are represented by Javascript values `1`, for `True`, or `0`, for `False`. To accomplish this, the code generator explicitly treats `Bool` values specially by specifically emitting `1` or `0` respectively, rather than `[2]` and `[1]` as is done for other algebraic types. This could be done for any type which only has argument-free data constructors by adding a bit more complexity to the code generator, but is currently only being done for `Bool`, to leverage Javascript's built-in concept of boolean logic.

4.1.7.2 Algebraic data types The representation of algebraic data types is based on arrays, as they offer faster lookups and more compact literal representation than objects, the other compound data type offered by Javascript. An algebraic value is represented by an array with a numeric tag indicating which constructor was used to create the value as its first element. This constructor tag is given by the STG source and, while STG gives no guarantee to that effect, always starts at 1, for the first data constructor of a given type, and increases by 1 for each consecutive constructor:

```
data SomeType
  = Foo -- Constructor tag: 1
  | Bar -- Constructor tag: 2
  | Baz -- Constructor tag: 3
```

This constructor tag is always a `Number`, and is followed by the each of the constructor's arguments, which may or may not be wrapped in thunks. A constructor's argument being wrapped in a thunk or not depends on the context of its instantiation; that is, Haste may decide to represent `(a, b)`[12] as either `[1,a,b]`, `[1,THUNK(a),THUNK(b)]`, `[1,THUNK(a),b]` or `[1,a,THUNK(b)]` in different places in the same program; the runtime system ensures that the generated code does not need to know or care that not all "thunks" are actually

[12]As we can see, tuples are not treated specially, in that `(a, b)` is equivalent to `(,) a b`, or *"the constructor tag for (,) followed by a and b"*.

represented as such; see sections 4.1.7.3 and 4.1.9 for more information about this.

Table 1 lists some common values and their simplest possible Javascript representations.[13]

`0.5 :: Double`	`[1, 0.5]`
`Nothing`	`[2]`
`[]`	`[2]`
`['a', 'b']`	`[1, [1, 'a'], [1, [1, 'b'], [2]]]`
`Left (0 :: Int)`	`[1, [1, 0]]`
`(True, False)`	`[1, 1, 0]`

Table 1: Representations for some common algebraic values

4.1.7.3 Thunks Thunks have slightly different needs than algebraic values; in particular, as Haste allows unthunked values to be used as thunks, it is important to distinguish thunks from values as fast as possible. To this end, Haste represents thunks as Javascript objects of the class `Thunk`, defined in the runtime system as:

```
{f: <if not evaluated then thunk body else 0>,
 x: <if evaluated then thunk value else undefined>}
```

That is, an object that has either a pointer to the function computing the thunk's value, if the thunk has not yet been evaluated, or a pointer to the value produced by calling the thunk's body function. To conserve memory, a thunk may not have both a body and a value at the same time.

Using a special class for thunks allows us to use the `instanceof` operator to distinguish thunks from values, which measurements have shown to be faster than array or object lookups in both Chrome's *V8* and Firefox's *SpiderMonkey* Javascript engines.

The creation and evaluation of thunks is described in further detail in section 4.1.9.

[13]As described above, all of a constructor's non-strict arguments are subject to thunking if Haste deems it necessary, meaning that all constructor arguments in table 1 may or may not be thunked in an actual program.

4.1.8 Foreign function interface

Haste supports importing functions from native Javascript code using the Foreign Function Interface extension. A Javascript function can easily be imported using FFI syntax:

```
foreign import ccall myFun :: Int -> IO Double
foreign import ccall myPure :: Int -> Int
```

The calling convention specified is irrelevant, as Haste ignores it. The FFI extension, however, demands that it exists. Although importing is easy, writing Haste-compatible Javascript functions requires some precautions.

As far as Haskell is concerned, the outside world is all about IO, and it treats all foreign imports as though they were IO computations. IO computations internally take one additional argument, representing the state of the world before the computation happens, and returns an extra value representing the state of the world after the computation has finished. In addition, function arguments of primitive types are automatically unboxed before being passed to a foreign function, and return values of primitive types are automatically boxed. Thus, a function that has the type `Int -> Int` in its `foreign import` declaration gets the type `Int# -> RealWorld# -> (# RealWorld#, Int# #)` as far as foreign functions are concerned.

The actual values of the `RealWorld` type are unimportant; they exist purely to ensure that foreign code is never memoized, as it may contain side effects, and are never examined. Even so, at present, Javascript functions written to be imported by Haste programs must take an extra parameter, and return an unboxed tuple with a `RealWorld` value, which may be anything, as its first element. Thus, if one wanted to make use of a Javascript-native `square` function, this is how it would be written:

```
-- Square.hs
module Square (square) where
foreign import ccall square :: Double -> Double
// NativeSquare.js
function square(x, _realWorld) {
        return [1, _realWorld, x*x];
    }
```

As can be seen in the return value of the native `square` function, the data constructor for an unboxed tuple is 1, and it's followed by the `RealWorld` parameter and the function's actual return value.

As the RealWorld parameter only exists for the benefit of the type system, its existence as far as external code is concerned is merely temporary. Native GHC removes this parameter during code generation, and Haste ought to as well, eventually. Similarly, the unboxed tuple return value could be optimized away, giving slightly faster code and less of a headache when writing Javascript bindings.

In this manner, it is possible to create foreign functions that work on all primitive Haskell types. In addition, using the information given about Haste's internal data representation in section 4.1.7, it is possible to write foreign functions that work with more complex data types as well by wrapping said types in a foreign pointer:

```
    -- SafeDiv.hs
    module SafeDiv (safeDiv) where
    import Haste.Prim (Ptr, toPtr, fromPtr)
    foreign import ccall safeDivJS :: Double
                                   -> Double
                                   -> Ptr (Maybe Double)
    safeDiv :: Double -> Double -> Maybe Double
    safeDiv a b = fromPtr (safeDivJS a b)
    // NativeSafeDiv.js
    function safeDivJS(a, b, _realWorld) {
        if(b == 0) {
            // [2] represents Nothing
            return [1, _realWorld, [2]];
        } else {
            // Create a Double using the D# constructor
            var result = [1, a / b];
            // Wrap the result in a Just constructor
            var justResult = [1, result];
            return [1, _realWorld, justResult];
        }
    }
```

As we can see, this example is considerably more complex. This is due to the fact that anything wrapped in a foreign pointer is treated as an opaque value; nothing is automatically unboxed for us except the outermost `Ptr` wrapping, and function arguments wrapped in a `Ptr` may even be thunks! Thus, `safeDivJS` must take care wrap its return value up in the proper constructors before returning. If it had had arguments wrapped in a `Ptr` as well, it would also have had to evaluate and unpack those arguments manually.

While it's safe to allow foreign functions to deal with arbitrary complex Haskell values without inspecting them, passing them around as opaque parcels, writing code like `safeDivJS`, which actually manipulates Haskell values, is *very* dangerous and generally discouraged, as it may not only violate data immutability by modifying its arguments, but also depends on the internal data representation staying constant, which is not guaranteed by any means.

While the Haste standard libraries make use of these technique to speed up some common high level functionality by implementing them in pure Javascript, code that is not closely coupled with the compiler should avoid them under all circumstances.

4.1.9 The run-time system

In order to conserve space execution time, the run-time system of Haste is very small; aside for primitive operations, it consists of three functions: *thunk*, *eval* and *apply*.

4.1.9.1 Thunk The *thunk* operation is very straightforward; it takes a nullary function as its argument, and creates a new `Thunk` object, introduced in section 4.1.7.3, with an undefined value property, using the given function as the thunk's body.

4.1.9.2 Eval Eval is slightly more complex. It takes a single argument, representing a supposed thunk. As Haste allows non-thunk values, such as `[1, 'x']` or `9.7`, to be used wherever a thunk is expected, behaving as though they were previously evaluated thunks, eval's first task is to determine whether its argument actually *is* a thunk. This is accomplished by checking whether its argument is an instance of the `Thunk` class or not. If it isn't, eval simply returns its argument. If the argument is indeed a thunk, eval then checks whether it is evaluated or not. If it is, its value is returned.

If the thunk is *not* evaluated, the next step is slightly different depending on whether the program was compiler with or without trampolining. Without trampolining, the thunk's body is called, the result assigned to its value property, its body component set to 0[14] to allow the thunk's now unnecessary body to be garbage collected, and its newly acquired value returned.

With trampolining, the thunk is evaluated repeatedly in a loop as long as evaluation returns a new thunk. Thus, a trampolining tail call may be performed by simply returning a thunk containing the function to be called.

4.1.9.3 Apply The most complex operation of the runtime system is the apply call, which handles function applications that Haste couldn't guarantee to always be saturated. As working with functions is relatively natural in Javascript whereas stack functionality is best handled implicitly using the call stack, using the eval/apply approach to implementing function calls rather than push/enter was a natural choice.[Marlow, Peyton Jones; 2004]

As calling a function using apply is quite a bit slower than the saturated function calls Javascript supports natively, Haste emits native function calls, or *fast calls* in Haste parlance, whenever possible. Still, there are cases when fast calls are not possible.

The arity of the callee is compared to the number of arguments provided, and appropriate; if the application is oversaturated, the callee is called with the appropriate arguments for its arity, and its return value is then assumed to be a function, and gets called with the remaining arguments; if the application is

[14]Measurements in both *V8* and *SpiderMonkey* show that carrying around a few bytes of extra memory by setting the body property to zero is significantly faster than deleting it outright.

undersaturated, a closure is returned, with the arity required to saturate the application[15]; if the application is fully saturated, the function is simply called as a normal Javascript function.

4.1.10 Libraries and output formats

Haste produces a *jsmod* file for each compiled module, written to the working directory of the compiler process. It also keeps its own cache of intermediate code for libraries installed with the `--libinstall` compiler flag. This code cache is usually kept in `~/.haste/lib` and contains a directory structure based on the fully qualified names of the modules it contains; the module `Data.Maybe` is found in `~/.haste/lib/Data/Maybe.jsmod` and so on.

The jsmod file format is not clearly defined as the code generator, and thus the information the format needs to accomodate, is still in flux; the *cereal-derive* package is used to automatically derive a binary file format, based on the algebraic data types representing the AST.

When compiling a complete application, as opposed to a library for installation into the code cache, hastec outputs a self-contained Javascript file named after the input file containing the program's Main module; if Main lives in the file *program.hs*, the resulting Javascript program will be written to *program.js*, unless otherwise configured using the command line parameter `--out=some_other_file.js`.

4.2 Base library

With the transition to a web-based execution environment, concepts like the standard output handle and local files disappear from the developer's toolbox. Instead, the need arises for a standard library that allows the programmer to react to user input, to manipulate items on a web page and to communicate with remote servers. Haste provides the means to accomplish all of these tasks.

[15]If a function with arity $n > m$ is called with m arguments, a closure with arity n-m will be returned.

4.2.1 The Fursuit library

The Fursuit library makes up the core of the libraries used by Haste programs to interface with the user. It does not make use of any Haste-exclusive features and lives in its own package, which the Haste package then depends on. Thus, it is perfectly possible to write reactive programs using Fursuit using vanilla GHC as well.

4.2.1.1 Library design The library's design is heavily influenced by [Elliott; 2009] and bases its interface on `Functor` and `Applicative` instances, with a small number of basic primitives and a larger number of derived combinators. One crucial difference from other FRP formulations, however, is that the behaviour and event concepts are merged into a single type: `Signal`. The reason for this is to provide a simpler interface, in particular for Javascript web developers without previous Haskell exposure. For the same reason, the notion of implicit time is not present in Fursuit, as it is rarely needed in a web context, where responding to discrete events is by far the most common use case, and can be adequately simulated by an explicit clock tick event, should the need arise. Signals are not allowed to recursively refer to themselves, as the semantics of doing so are not obvious, making programs harder to write and understand.

Fursuit does not expose a monad interface, in part to avoid time leaks, as described in [Apfelmus; May 2011], and in part to enable a simple and efficient implementation, as described in section 4.2.1.2.

A signal is said to *fire*, *update* or "change its value" whenever an update occurs in a signal that it depends on; this happens regardless of whether the signal actually changes or not. Thus, a signal that always has the value of `''foo''` may still fire and cause the signals to depend on it to fire as well, in spite of always having the same value.

Sinks In order to make these signals actually do something, the concept of a *sink* is used. A sink is simply a callback that is called with the value of a signal whenever that signal fires, and created using the function `sink :: (a ->`

`IO ()) -> Signal a -> IO ()`, the usage of which should be fairly evident from its type. A sink created in this manner is guaranteed to be called exactly once for each firing of its input signal. Fursuit also exposes a type class `Sink s a`, with a single method: `(<<) :: s -> Signal a -> IO ()`. This method allows programmers to implement the sink concept in a more concise way. For example, to update an `IORef a` whenever a certain `Signal a` fires, one would simply write `theRef << theSignal`, making the intended semantics of the operator slightly like Unix output handle redirection.

The primitives Fursuit has the following primitive operations:

- *Lifting*, which lifts a value to a signal. This is expressed as the `pure` method of `Applicative`.
- *Application*, which applies a lifted function contained in a signal to a value also contained in a signal. This is the `<*>` method of `Applicative`.
- *Pipe*; a signal that changes its value whenever a new value is written to the pipe's corresponding input handle. This is the only way to inject a value into a signal network, and is used to implement events. A pipe is created using the pipe :: `a -> IO (Pipe a, Signal a)` function, which takes an initial value for the pipe and creates a `Pipe` value, that represents the writing end of the pipe and may be written to using the `write :: Pipe a -> a -> IO ()` function, and a signal that represents the reading end of the pipe, and changes its value whenever something is written into the pipe's writing end.
- *Filtering*, which "removes" events that don't match a predicate. For instance, a signal that may not take on a negative value may be expressed as `filterS (>= 0) signal`.
- *Union*, which merges to signals into one. To create a signal that changes its value to whichever of sig1 and sig2 last had its value updated, one would write `sig1 ‘union‘ sig2`. The union operation is left-biased, meaning that if `sig1` and `sig2` fire at the exact same instant, `sig1` is preferred over `sig2`.

- *Accumulation* is probably the most complex primitive. It takes the form of the function `accumS :: a -> Signal (a -> a) -> Signal a`; the first argument is the signal's initial value, and the second is a signal containing a function. Whenever the function signal fires, the function contained therein is applied to the last value of the accumulator signal, yielding a new output value. This is the primary way of implementing state within a signal.

- *Impure signal instantiation* is not as much a primitive as a convenience function implemented as a primitive. Sometimes it's necessary to create signals impurely; for instance, when setting up signal that corresponds to the value of a text field on a web page. Often, these signals are only used once, in which case adding a whole new line just to express `sig <- createSomeSignal` just adds unnecessary clutter. To do away with this clutter, the function `new :: IO (Signal a) -> Signal a` can be used to create this signal in-line. Its semantics mimic those of the `new` keyword of C++ or Java, meaning that each occurrence of the `new` function only ever creates a single signal, ensuring that the signal is not created multiple times in the case the user does something like `let sig = new mkSig in sig ‘union‘ sig`; GHC does not inline `sig` in this expression, taking care not to break the sharing we just introduced. Of course, `new mkSig ‘union‘ new mkSig` still creates two signals, as expected. This single signal is created when the first signal referring to it is attached to a sink.

Derived combinators In addition to these primitives, a set of derived combinators are provided for convenience.

- `unions :: [Signal a] -> Signal a` extends the concept of signal union to an arbitrary number of signals. As it is based on the `union` function, it is still left biased.

- `stateful :: (st, a) -> Signal (st -> (st, a)) -> Signal a` wraps the accumS primitive into a form which more explicitly keeps state and returns values separate from that state.

- `filterMapS :: (a -> Maybe b) -> Signal a -> Signal b` combines the functionality of the filterS primitive and function application; any value for which the given function returns `Nothing` is stopped dead in its tracks.

- `whenS :: Signal Bool -> Signal a -> Signal a` extends the concept of filtering to allow a changing predicate; firings of the value signal propagate if and only if the predicate signal holds.

- `zipS :: Signal a -> Signal b -> Signal (a, b)` performs the same functionality for signals that `zip` performs for values.

- `fromS` and `untilS :: (a -> Bool) -> Signal a -> Signal b -> Signal a` are signals that propagate their value signal from the point in time where their predicate first holds, or until the point in time where it first does not hold, respectively. The third argument is a reset signal, allowing them to "forget" their predicate's history of being true or false.

- The `Functor` and `Applicative` type classes provide their own set of derived combinators. In particular, the <$ operation of applicative is very useful for causing a signal to change its value to something unrelated to its signal input. For instance, the signal `''something happened'' <$ sig` will fire an event and take on the value of "something happened" whenever the signal `sig` fires.

4.2.1.2 Under the hood As Fursuit's main interface are the `Applicative` and `Functor` type classes, implementation is relatively straightforward. Signal uses deep embedding; it generates a data type representing an abstract syntax tree from its combinators, which is then compiled and actuated using the `sink` function.

The `sink` function first traverses the generated data type in order to find all pipes which affect the value of the signal, adding the compiled signal as a *listener* to each pipe. Whenever a value is written to a pipe, each of its listeners are notified, causing the initial reaction to events to be push-based, whereas the actual response is pull-based. This design, while possible less efficient and elegant than *reactive-banana* and *reactive*, has the great advantage of being feasible without any concurrency support at all - this is crucial for its use in a the non-concurrent Javascript environment Haste has to deal with.

Not using a monadic interface is crucial for this traversal, as the structure of a monadic computation can not be known beforehand due to the type of the bind operator, making this traversal very difficult. This is also the reason for choosing deep embedding; in a shallow setting, `Signal` values would simply consist of opaque functions, which would make it impossible to trace values back to their sources.

When the response to a signal firing is computed, the *origin path flag* for each signal primitive is calculated; the origin path starts at the pipe that triggered the signal firing, and each primitive that depends on a signal which is on the origin path, is also on the origin path. This flag is used to determine whether to return the previourly cached response for an `accumS` primitive, which will yield incorrect results if its function signal is applied in spite of not being on the origin path. Ponder, for example, the following code:

```
(p1, updater) <- pipe (+1)
(p2, unrelatedSignal) <- pipe “foo”
sink print (zipS (unrelatedSignal, accumS 0 updater))
write p2 “bar”
write p2 “baz”
```

This creates a `Signal (String, Integer)` and binds it to a sink that prints the value of its input signal whenever it fires. As `unrelatedSignal` is not on the origin path of the `accumS` part, the two writes to p2 will result in the printing of `(“bar”, 0)` and `(“baz”, 0)` respectively, as the `accumS` primitive will see that its input did not change, and that its previous state should therefore be returned. Had we not had the origin path flag, `accumS` would have had no way to know that its input function - the `(+1)` given as the initial value of the `updater` signal - did not change, instead applying it to its state for each write to p2, instead printing `(“bar”, 1)` and `(“baz”, 2)` - clearly not what we wanted or expected!

Currently, the origin path flag calculation is coupled with the actual result calculation. In the future, decoupling this calculation from the result may allow for all signals to cache their values based on this flag, cutting down on unnecessary recomputations for complex compound signals.

4.2.2 Basic DOM and AJAX facilities

While the reactive core of Fursuit is a good start, without bindings to the web browser, it doesn't actually do anything. Haste provides a small library for causing side effects within the browser.

4.2.2.1 Imperative building blocks At the bottom of these libraries lies the `Haste.Prim` module, providing the basic functionality for converting between Javascript strings and Haskell's character list strings and working with the FFI, as described in section 4.1.8. It also contains efficient conversions between Haskell's numeric types, as well as their conversions to and from `String`, as these are simple primitive operations in Javascript but comparatively bulky and slow in Haskell's base libraries.

On top of this sits the basic DOM library, which provides primitives for locating DOM elements by their ID, getting and setting their CSS styles and properties, and enumerating, clearing, setting, adding and removing their child elements. It also provides functionality for attaching IO actions as callbacks for most DOM events, including but not limited to mouse clicks, keypresses, changes in input controls such as text fields and check boxes, and timeouts.

Alongside the DOM library lives the AJAX library, which provides the ability to perform AJAX requests to remote servers, either as plain text or using the JSON format. The JSON parser and serializer are implemented entirely in Javascript, called via the FFI bindings; while this is more complex than implementing the functionality in Haskell, not only because Javascript is a lower level language than Haskell but also because the Javascript JSON functions need to deal intimately with Haste's internal data representation, JSON requests are common enough that execution time and code footprint become important considerations.

4.2.2.2 Reactive bindings The libraries above have all been thin wrappers around native, imperative, Javascript functionality. These building blocks are used in conjunction with Fursuit to provide a reactive interface to events and AJAX requests.

The module `Haste.Reactive.DOM` defines basic functionality for working reactively with DOM elements. In particular, functions exist to set up signals detecting mouse clicks, state changes and other events coming from the DOM. The module also defines `Sink` instances for DOM elements and their properties, allowing the programmer to replace an element's children or change its properties reactively using the << operator. For instance, this short code snipped keeps a DOM element with the ID "myLabel" updated with the reverse of any text typed into a text box with the ID "myTextbox":

```
elemProp ‘‘myLabel.innerHTML’’
    << reverse <$> new (‘‘myTextbox’’ ‘valueAt‘ OnKeyUp)
```

Meanwhile, this program displays a message box with the number of times the user has clicked the button "myBtn":

```
(alert . show_) << accumS 0 ((+1) <$ new (clicked ‘‘myBtn’’))
```

Note the use of the <$ operator to trigger a signal to fire with the value of `(+1)` whenever a signal completely unrelated to that value fires.

The module `Haste.Reactive.Ajax` defines functions for creating signals based on AJAX calls. When an input signal to an AJAX signal is fired, the request associated with the signal is dispatched. The signal then goes to sleep until a response to the request arrives, at which time the signal continues propagating. AJAX signals are merely a convenience combinator, and could be simulated using an extra sink, which would receive the event that triggers the AJAX request and dispatch it, firing another completion signal when the response arrives.

The following program will fetch and display a weather forecast from a server whenever the user types a city name into a text field and presses return:

```
    thisCity <- valueOf “thisCity”
    let cityArg = fmap (:[]) (zipS “city” thisCity)
    alert << ajaxSig (pure “/weather_forecast.cgi”) cityArg
```

A request is made to the CGI program `/weather_forecast.cgi`, with the query string `?city=<what the user typed in the textbox>`. The `zipS` combines “`city`” with the name of the city read from the text box into a `Signal (String, String)`, and applying `fmap (:[])` to the result turns it into a `Signal [(String, String)]` - a signal representing the (key, value) pairs making up the query string of the URL.

For examples of how these signals could be combined into larger programs, the Haskell port of the *Glosie* program is a good starting point.[Ekblad; 2012]

4.3 Performance measurements

The main three performance metrics of the code generator are source code size, output code size and execution speed, with a heavy emphasis on the two former. While high speed execution is an explicit non-goal of Haste, practical deployability taking precedence, it is nevertheless important to assess whether the generated code executes *fast enough*; it does not matter how elegant, correct and well-typed your code is if it is too slow to drive an interactive web page without annoying the user with long pauses and lagging input. Fast enough being a somewhat subjective term, this section limits itself to comparisons with native Javascript, GHCJS and UHC, being the three main alternatives in the field. Output code size will be compared relative all three alternatives. Runtime performance will only be compared with that of native Javascript and UHC, as numerous time-consuming attempts of getting GHCJS to produce actual executable output have all failed. Source code size will only be compared to native Javascript, as this metric can be assumed to be fairly similar across the three Haskell compilers.

All figures relating to these performance metrics are presented relative to the baseline; how the native Javascript version of each benchmark performs. For instance, an entry having an execution time of 1.2 indicates that the entry is 20% slower than the Javascript entry. For all benchmarks, the Haste entry was compiled with `hastec -O2 --opt-vague-ints --opt-google-closure`,[16] the UHC entry with `uhc -O2 -tjs` and the GHCJS entry with `ghc --make -O2`, those being the flags supposedly producing the fastest and smallest code for each compiler.

4.3.1 Execution time

To get a rough idea about how well code compiled with Haste will perform relative to native Javascript and other Haskell to Javascript compilers, two microbenchmarks were used; *fibs*, which computes the 30th Fibonacci number recursively, and *reverse*, which reverses a linear data structure containing 100 000 elements. These two programs were implemented in Haskell and Javascript respectively, and their run times, averaged over 20 runs using the standalone version of Mozilla's SpiderMonkey Javascript interpreter, measured using the native Javascript version as the baseline. Note that the native version of *reverse* creates and reverses an array while the Haskell implementation uses a list, the two being the basic primitive data structures of their respective languages.

As we can see from the chart, UHC tends to be about two orders of magnitude slower than native Javascript, requiring a logarithmic scale for the chart to be readable. Haste, quite surprisingly, actually beats the native implementation by a margin of about 10% for the *fibs* benchmark, but is still roughly 7.5 times slower for the *reverse* case.

Examining the code generated by Haste, GHC has managed to strip away all traces of laziness for *fibs*, allowing the code generator to simplify the output to something very similar to the native version. For *reverse*, the Haste entry is basically doing nothing but creating and evaluating thunks in a loop, taking this benchmark rather close to its absolute worst case, performance-wise.

Overall, while microbenchmarks are infamously bad at accurately predicting the performance of real applications, these numbers at least hint that a program compiled with Haste will rarely perform more than an order of magnitude worse than an equivalent application writtne in native Javascript.

[16] `--opt-vague-ints` turns off the strict checks that integer arithmetic is always performed modulo 2^{32}, reducing the execution time of the integer-heavy *fibs* microbenchmark by about 4%.

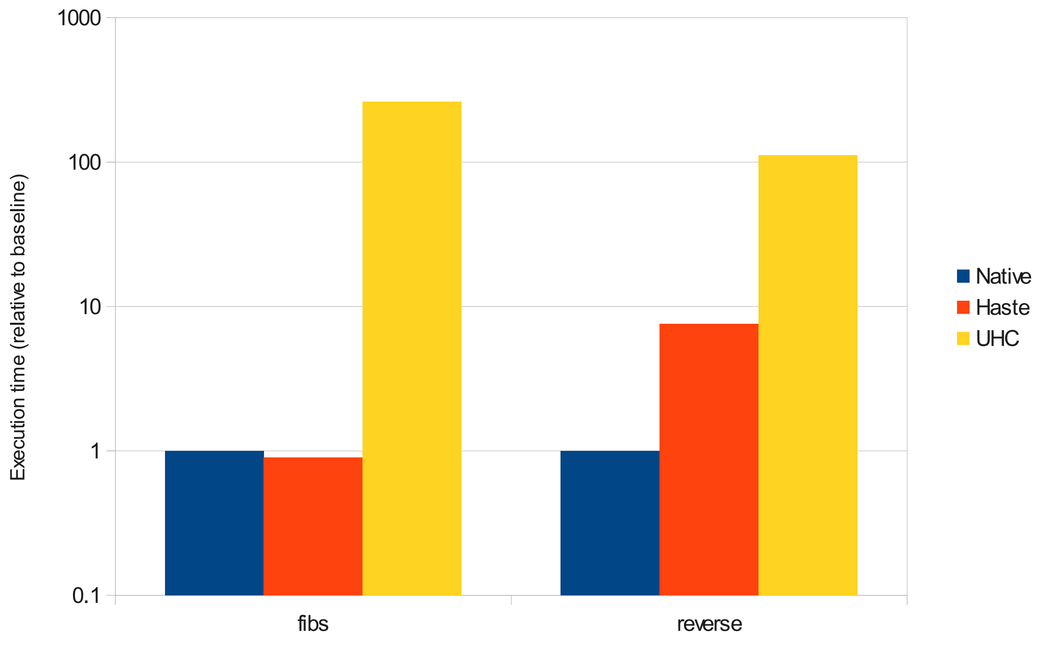

Figure 2: Execution times relative to baselime

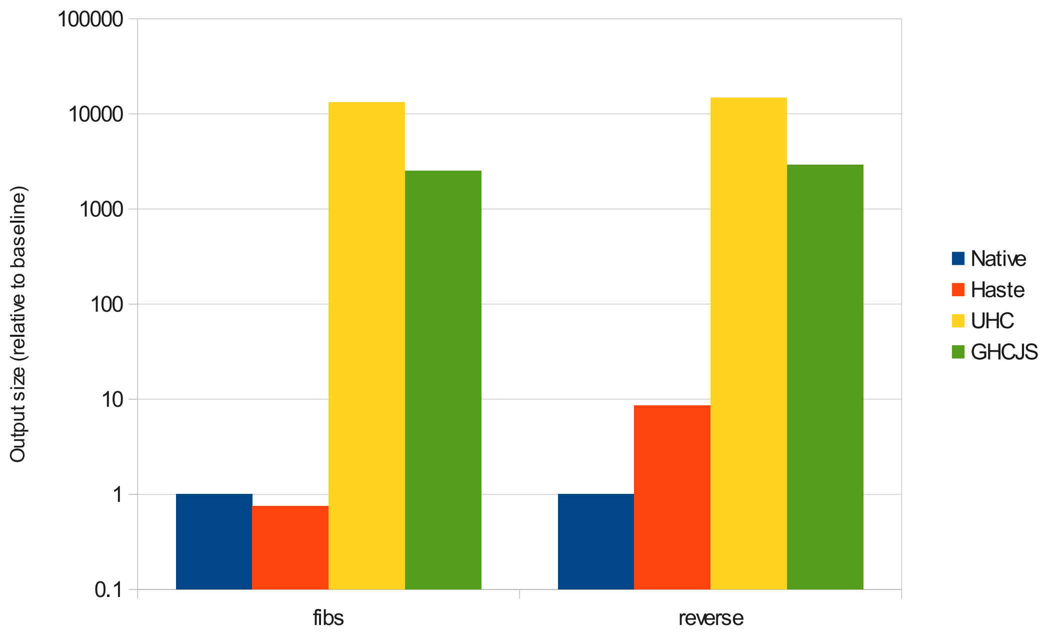

Figure 3: Output code size, relative to baseline

4.3.2 Output code size

The primary obstacle to deployability Javascript code generated from a higher level language is probably the size of the output code; while computing power is relatively abundant even in low-end smartphones, transferring a multi-megabyte Javascript blob over a mobile UMTS link before a web application can be used makes for a horrible user experience. Thus, it is critical for Haste to produce relatively lean code if it is to be at all useful.

This benchmark simply compares the size of the code output produced by compiling the two microbenchmarks, *fibs* and *reverse*, using each of the three compilers to the size of the native Javascript implementation of each. In the case of GHCJS, the comparison is not entirely apples to apples since it does not include all necessary dependencies for the code it generates to actually be executable, artificially deflating its output code size.

Looking at this chart, Haste again does relatively well, even beating the native implementation by a small margin for the *fibs* program, with the *reverse* program being almost an order of magnitude larger. That the laziness-heavy *reverse* produces larger code comes as no surprise; the verbose Javascript syntax for anonymous functions adds an overhead of at least 18 bytes per thunk creation site. Considering that the native *reverse* program is only 127 bytes in total, it is easy to see why removing laziness wherever possible is important for minimizing code size.

It is important to note, however, that microbenchmarks are a relatively poor indicator of the size of actual applications. In particular, UHC does not seem to do any function level linking, thus including quite a lot of code that is never actually used. Obviously, this incurs a huge penalty for a simple microbenchmark such as this, but the effect gets progressively less pronounced as the complexity of an application grows. Thus, compiling the Haskell equivalent of a 2 MB Javascript program with UHC would hardly yield 20 gigabytes of code.

4.3.3 Source code size

While examining small benchmark programs may be useful for an apples to apples comparison with existing efforts, when comparing with native Javascript, it is more interesting to look at a larger code base, that actually does something useful. For the sake of this comparison, the [Skårstedt; 2012] simple glossary training web application, written by a professional web developer, was rewritten in Haskell, using Haste's reactive standard library, and the sizes of the two resulting code bases were then compared. In addition, the size of the Haste-compiled output of the Haskell version was compared to the size of the original application, to provide a very rough estimate of how code size differs between Haste and Javascript for slightly more complex examples than the previous microbenchmarks. Unfortunately, as the Haskell version of the application depends on libraries only available in Haste, no comparison could be done with GHCJS and UHC.

Again, the Javascript version was chosen as the baseline. For the source code, we see that the Haskell program is less than one fourth of the size of

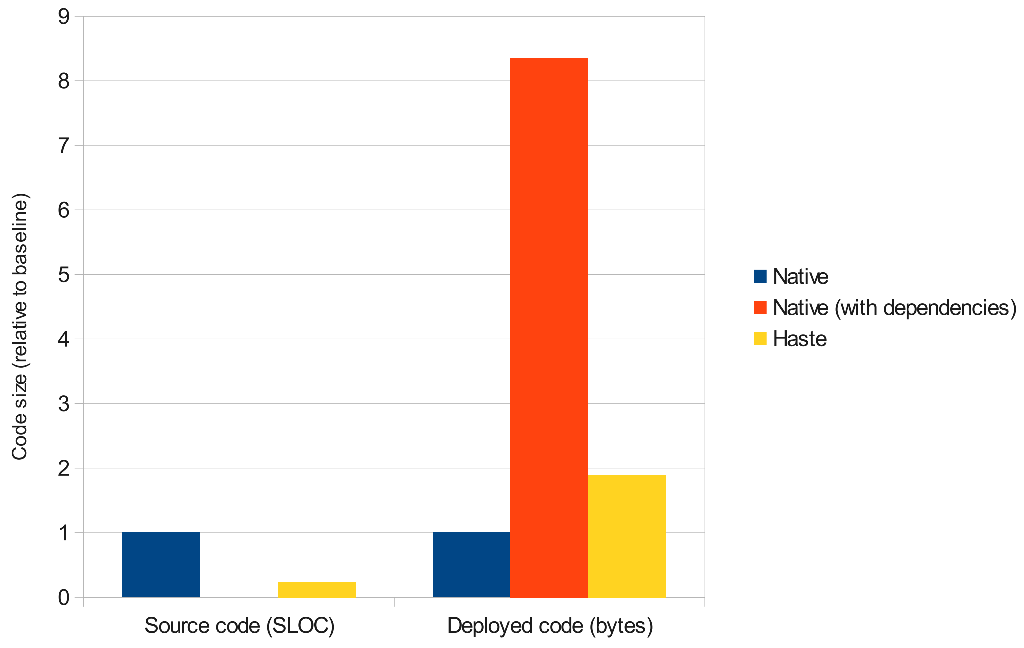

Figure 4: Source and deployed code sizes for the two Glosie versions

the original Javascript application, counting lines of code. For the output, the Haskell version turns out to be nearly twice as large as the original. There is a twist here, however; the Haskell version is a self-contained Javascript blob, whereas the original depends on the presence of the [jQuery] library to run. While this may still be a fair comparison, considering that jQuery is commonly distributed from a single shared source and cached locally by web browsers, instantly available to any web application that requires it, if we were to include the size of jQuery in the code footprint of the original application, it suddenly becomes four times larger than the Haskell version. As jQuery is distinctly separate from any program making use of it and is always delivered as a tightly compressed blob lacking any whitespace, including it in the source code size comparison is not meaningful. Similarly, the line count of the base libraries are not included in Haste's source code size.

5 Notable problems

While Haste represents a significant step forward over Javascript, there are some areas of its implementation that prove fairly problematic. Javascript's somewhat eccentric feature set rears its ugly head not only when used as a high level language, but also when used as a compilation target; in some cases, even more esoteric features enables such quirkiness to be papered over with a little effort, while some cases prove to be quite the challenge, with no obviously "right" solution.

5.1 Bitwise operations on 32-bit Words

When serving as the operand of a bitwise operation, a Javascript Number behaves like a 32-bit signed integer. For instance, `Math.pow(2, 31) & 0xffffffff` becomes -2147483648. This behaviour can be quite a problem, as it is rather common to want to interpret the operands and result of a bitwise operation as unsigned values rather than signed. Fortunately, the logic right shift operator, >>>, interprets its result as a 32-bit unsigned value, allowing for correct implementation of the unsigned `Word` type by simply logically shifting the result of any bitwise operation on unsigned values to the right by 0 bits.

5.2 Tail recursion and by-value closures

When optimizing tail recursion into loops, a problem with Javascript's closure semantics rears its ugly head; more specifically, the fact that all variables captured by a closure are captured by reference. Ponder the following function:

```
tailsum [] accum     = accum
tailsum (x:xs) accum = tailsum xs (x+accum)
```

Without any optimizations, it will compile into the following pseudo-Javascript:

```
function tailsum(xs, accum) {
  if(EMPTY_LIST(xs)){
    return accum;
  } else {
    var accum2 = THUNK(function(){
                   return EVAL(HEAD(xs)) + EVAL(accum);
                 });
    return tailsum(TAIL(xs), accum2);
  }
}
```

In the inductive case, a thunk is built up that, when evaluated, adds the head of the list to the accumulator, and then passed to the next call to `tailsum`. This is perfectly fine; since variables have function scope, each new call to `tailsum` will create new variables `x` and `accum` for the thunk to capture. However, when we compile tail recursion into loops, we get this function instead:

```
function tailsum(xs, accum) {
  while(true) {
    if(EMPTY_LIST(xs)){
      return accum;
    } else {
            accum = THUNK(function(){
                      return EVAL(HEAD(xs)) + EVAL(accum);
                    });
            xs = TAIL(xs);
            continue;
          }
        }
      }
```

Now, instead of calling itself, `tailsum` uses the `continue` statement to jump back to the beginning of the loop, after assigning new values to `xs` and `accum`. Unfortunately, this innocent-looking optimization has broken our program.

Recall that when Javascript creates a closure, all captured variables are captured by reference. Thus, when we overwrite a variable, any closure referring to that variable will now see its new value rather than the old one, as we wanted. In this particular example, when evaluated, the thunk assigned to `accum` will attempt to evaluate itself when it reaches `EVAL(accum)`, in the mistaken belief that `accum` refers to its old value, serting in motion an infinite recursion that can only end with the exhaustion of the call stack and the subsequent demise of our program.

To combat this problem, a trick that simulates by-value closures is used, called *const closures* in the AST. To create one of these, we wrap every closure in an anonymous function with every captured variable as arguments, to create an inner scope for our closure, which is then called with the same variables from the outer scope as arguments. With this workaround, this is what the `accum` closure from our example will look like:

```
      accum = (function(xs, accum) {
                   return THUNK(function(){
                             return EVAL(HEAD(xs)) + EVAL(accum);
                           });
               )(xs, accum);
```

As we can see, the variables `xs` and `accum` from the outer scope, is copied into an inner scope by the function call, where they are subsequently captured by the thunk.

This more expensive way of creating closures is employed in all tail looping functions in order to preserve correctness, the list of variables to copy generously provided by the list of free variables attached to every STG closure. This overhead occurs only when creating closures, not when calling them, making their performance impact negligible in the face of the improvements gained by optimizing tail recursion into loops in the first place.

The observant reader may notice that this trick relies on the order of assignment of the tail looping function's arguments; if `xs` were to be assigned before `accum`, `accum` would still get the wrong value for `xs`, which would be even worse, as we would then get a program that doesn't crash, but silently returns the wrong result! In the actual code generated by Haste, however, the assignment of the loop arguments happen in two steps; the first assigning the arguments to temporary variables, the second using those temporary variables to assign all of the loop arguments at the same time, preventing this from being a problem. In reality, the loop assignments would look like this rather than the simplified version in the example:

```
var tmp1 = TAIL(xs);
var tmp2 = (function(xs, accum) {
              return THUNK(function(){
                       return EVAL(HEAD(xs)) + EVAL(accum);
                     });
           )(xs, accum);
xs = tmp1;
accum = tmp2;
```

5.3 Problematic GHC assumptions

As GHC is first and foremost a compiler from Haskell to one of several native machine architectures, it makes some assumptions that sometimes get in the way of using it as a library for what is essentially a cross compiler. In particular, it assumes that the build platform and the target system have the same word size; an entirely reasonably assumption to make when you're not a cross compiler, but slightly problematic when aiming to be a more general Haskell compilation library. This is not so bad on its own, as [Peyton Jones et al; 2002] explicitly states that the only guarantee made about the range of the `Int` type is that it will be signed and have at least 30 bits of precision, which enables Haste to simply truncate all `Int` values to 32 bits, optionally printing a warning whenever this truncation occurs.

What really makes this a problem, however, is that GHC also assumes that if the host system has a word size of n bits, then it's perfectly valid to coerce values between `Int` and `Int`*n*! This is quite unfortunate, as the code generator needs to know whether any given symbol is definitely intended to have 64 bits of precision, requiring special measures as there is no way to natively represent a 64 bit integer in Javascript without losing precision, or if it should just happily perform the aforementioned truncation and be done with it.

One possible solution to this problem would be to harness the plugin system introduced with GHC 7.2. A plugin module could be written which, when executed at the very beginning of GHC's optimization pipeline, would traverse the Core of each module, wrapping each instance of `Int64` and `Int32` up in an `Opaque_Int64/32` type with its own distinct instances of the numeric type classes, ensuring that GHC does not perform any optimizations that get in the way of Haste's code generator.

This is as yet purely theoretical, however, and further research would be needed to implement and verify its efficacy.

5.4 64-bit integers and 32-bit multiplication

Due to double precision IEEE754 floating point numbers being the only basic numeric type available in Javascript, operations on 64-bit integer data types are non-trivial to implement. Doubles have a 53 bit mantissa and an 11 bit exponent, which allows us to perform arithmetic with operands and results in the range [-2^{52}, 2^{52}-1] without losing precision. Not only does this mean that natively representing 64-bit integers is impossible, but also that multiplying two large 32 bit integers may give the wrong result. At present, Haste does not take any measures to solve either problem, as there are problems associated with each candidate solution.

For the former, there are several solutions:

- Use a 64-bit integer type, such as the `google.math.Long` class implemented for the Google Web Toolkit, for the 64 bit types. Unfortunately, as described in section 5.3, GHC freely converts between `Int` and `Int64` on a 64-bit system, making it very hard to distinguish which values were meant by the programmer to be of a 64-bit type.

- Use a 64-bit integer type for *all* integers, essentially telling GHC that it's compiling for a 64-bit architecture. This brings problems for 32-bit systems instead, as GHC shamelessly treats `Int` and `Int32` as the same thing on such systems, the converse of its 64-bit counterpart. This may also seem unreasonably costly for the vast majority of operations that do not need 64 bits of precision.

- Use an arbitrary precision integer type for all integers, and perform all arithmetic modulo 2^{32} or 2^{64} as appropriate. This would have the added advantage of allowing all integral types to share the same arithmetic code. Again, the problem is to distinguish which integers are supposed to be what size, without which generating correct code would be impossible. In addition, this would be even more expensive than using 64 bits for all finite precision integers.

- Let the word size of the compiler's host machine decide whether to use 32 or 64 bits for integers. This solution has the advantage of actually being relatively easy to implement without conflicting with any GHC optimizations. However, as the compiler's target system is the same regardless of its host system, producing code with different semantics and performance characteristics depending on whether a developer uses a 32 or 64-bit operating system is unsightly, and definitely not something one would expect of a cross compiler.

In essence, the *raison d'etre* of 64-bit integer types is to allow arithmetic to be performed on larger values without the added overhead of an infinite precision integer library. Since their implementation in Javascript would perform on about the same level as a native infinite precision integer library, using `Integer` wherever `Int64` would otherwise have been used is definitely recommended, relegating 64-bit data types to a hack for compatibility with libraries making use of such types. Nevertheless, library compatibility being fairly important, combining the plugin-based solution speculated upon in section 5.3 with one of the solutions to the 64-bit problem is clearly a path worth exploring further.

For the latter problem, the textbook solution would be to split the operands into 16-bit chunks and perform separate multiplications on those, which was deemed too costly and time-consuming, in particular since it is not clear that such a solution would not be subsumed by one of the solutions to the general problem with 64-bit arithmetic, wasting any work put into solving this subproblem.

5.5 Redundant low-level operations

The Haskell standard libraries are written with a low-level native environment in mind. As such, when used by Haste applications, they redundantly implement quite a lot of features that already exist built into the Javascript engine or browser libraries Not only does this add unnecessary bulk to Haste programs, but also performs significantly worse than a thin wrapper on top of the browser's native functionality would. For example, a test program containing a single application of Prelude's `Show` instance for `Double` values produces nearly 25 kilobytes of Javascript and takes 25 milliseconds to execute. Compared to the 240 bytes of space and less than one millisecond of execution time required by an example program which uses Haste's `show_`, a thin Haskell wrapper over Javascript's `toString()` functionality, but is otherwise identical, this is evidently a real problem.

Unfortunately, replacing these operations with Javascript wrappers is not as simple as dropping modified intermediate code for the *base* package into Haste's library directory. GHC performs very aggressive inlining across module boundaries, as described in [GHC performance; Haskell Wiki], meaning that the low-level code ends up in quite a lot of libraries other than the ones where it was defined. Worse, when inlining, the interfaces of modules may change, and the code to inline is taken from GHC's own cache of intermediate Core, meaning that even if we used modified Haste intermediate code for every library on the system, in the best case, GHC would *still* inline that redundant and inefficient code; more likely, our program would simply not run due to the differences between the interfaces of the intermediate Core and Haste AST!

In order to escape this problem, we either need to use our own packages, e.g. *base-haste* instead of *haste*, or say goodbye to GHC's package database entirely. The first solution is not particularly appealing; it would make us unable to compile any library without altering their dependencies which, even if it could be done automatically, would be quite inelegant. The second solution would entail distributing a custom version of cabal, something the GHCJS developers, are currently exploring, which looks like a promising area of research to tackle this problem. This would also bring some marked improvements in other areas of library management, as detailed in section 6.2.

6 Future work

Although this thesis, hopefully, makes a sizable contribution to getting Haskell into every user's web browser, the path there is still long and filled with stumbling blocks, the journey only begun. There is much that can be done to remove the rough edges of Haste, both for developers and for the users of their code. While the problems described in section 5 mark the more immediate direction of development for Haste and thus consist future work in their own right, there are other areas that, while not being quite as urgent or closely tied to Haste itself, would do much to improve on the current situation.

6.1 A better standard library

A higher level standard library would do much to improve usability; while the current library already raises the abstraction level over what web developers deal with today, we can still do better. While the reactive core indeed eases a lot of the burden of handling events and input in interactive applications and the basic building blocks of the present DOM API serve rater well for what they are, an API that allows the developer to manipulate HTML elements and their properties as values and build even higher level GUI abstractions atop those, rather than passing around references to actual DOM nodes, manipulated by rather thin wrappers over the standard DOM manipulation functionality would be a huge improvement.

6.2 Keeping track of those libraries

Library bookkeeping and versioning is presently a problem. While the user “only” has to `cabal fetch` their desired libraries and build their exported modules with `hastec --libinstall`, this is cumbersome, prone to harmless but annoying mistakes such as forgetting to build one exposed module or another, and has to be repeated every time a library is upgraded; it’s simply a horrible user experience. For GHC proper, *cabal* already handles all of this in an adequate, if at times quirky and annoying, manner. One solution to this problem, currently employed by GHCJS as mentioned in section 5.3, would be to distribute a custom version of *cabal*, patched to build and install libraries for Haste into a separate package repository. This would also be a help better attuning the base libraries to the browser environment they’re ultimately going to execute in.

6.3 Libraries for Javascript

Haste currently only supports producing standalone Javascript applications. It might be useful, however, to be able to write libraries in Haskell which are then callable from native Javascript code.

6.4 Richly typed, safe RPC

As there is a plethora of server side web development frameworks for Haskell, it would make sense to attempt to achieve some sort of optional tighter coupling between Haskell clients and servers, to enable passing messages between the two with richer, statically checked types not available when using more widespread and general message formats such as JSON or XML.

6.5 Exploring recursive Signals

Unlike normal functions, the meaning of a reactive Signal referring to itself is not obvious. Does it imply recursion, or is it perhaps a reference to the Signal's previous value? Still, recursive signals would allow a greater degree of expressivity, in particular for the temporal combinators `fromS` and `untilS`. An interesting area of research would be to explore and formulate intuitive semantics for implementing recursive signals in the Fursuit library, increasing its expressive power while preserving its simplicity.

6.6 Better caching of inactive Signals

With the current Fursuit implementation, values are sometimes recomputed needlessly. An improved implementation could make more aggressive use of caching to avoid recomputing any values that can be proved to not actually have changed, based on which event sources triggered the event firing.

6.7 A modular, cross-compiling GHC

GHC is presently quite closely tied to the hardware it is built on. While this works well for a standalone compiler, if GHC aspires to be used as a library in other Haskell-related applications, this is less than optimal. One could envision a more modular GHC, where backends are separately installable and can be switched between by passing a command line flag or similar, allowing the same build of the GHC binary and library to produce code for a wide range of architectures.

7 Conclusions

This thesis set out to explore the feasibility of using Haskell to develop client side web applications and, more specifically, whether GHC rises to the challenge to be used as part of a cross compiler. While there had been several previous attacks on the problem, none could really be said to be practical; code footprints were large, execution speed abysmal, important features were lacking and compilation processes often came with disclaimers of “some assembly required” or “batteries not included”.

As shown in section 4.3, it is indeed possible to create programs with competitive execution speed and code footprint using Haskell. By leveraging state of the art tools such as GHC and the Closure compiler, Haste even outperforms native Javascript by a small margin in comparisons of both execution speed and code footprint under some circumstances, and produces code at least an order of magnitude faster and smaller than previous attacks in all conducted tests. Despite its primary function as a standalone native Haskell compiler, GHC sports a remarkable versatility with regards to its APIs and intermediate formats, without which creating practical client side web applications using Haskell may very well have been impossible without the investment of several person years worth of work.

Everything is not perfection and sunshine, however; as we have seen in section 5, problems still remain, in particular with regards to 64-bit data types. While its base libraries work very well for a native environment, the browser being a much higher level platform than any native compilation target makes many of its most basic components unnecessarily verbose an dinefficient. As GHC is first and foremost a native compiler, it would seem that further divergence, in particular with regards to library management, keeping a separate Javascript-specific package cache, is the way forward, as it would go a long way towards solving most of the problems encountered.

The base libraries for DOM manipulation, while workable and marking the beginning of a good foundation for higher level libraries, would have benefitted greatly from more attention; due to time constraints put on by the need to develop a completely new code generator, these libraries did not receive the time needed to make them truly shine. The build process may also benefit from splitting out the Haste-specific base libraries into a third package, enabling the entire project to be built and managed using Cabal alone, whereas it currently uses a Unix shell script to perform parts of the installation process.

Looking back at the goals set forth in section 1.4, however, the outcome of this thesis is undeniably favorable; Haste meets, and even exceeds, its performance goals, and is relatively easy to use compared to other attacks on the problem, using the same command line arguments as GHC and producing a single, self-contained Javascript file at the end of compilation.

While the problems with libraries and 64-bit types need fixing in order to turn Haste into a production level compiler, and there is a lot of polish that can and should be added to improve it even further as described in section 6, Haste represents a large step towards being able to use Haskell for the entire software stack making up a web application.

Haste can be downloaded from http://ekblad.cc/haste and be built by following the instructions in the included `README.md` file. The source code is documented using Haddock markup, meaning that HTML documentation can be built for both Fursuit and the Haste-specific base libraries by executing `cabal haddock` in the top directory of their respective source trees.

References

[Apfelmus; March 2011] Reactive-banana and the essence of FRP, http://apfelmus.nfshost.com/blog/2011/03/28-essence-frp.html (fetched 2012-06-01)

[Apfelmus; May 2011] FRP - Dynamic Event Switching, http://apfelmus.nfshost.com/blog/2011/05/15-frp-dynamic-event-switching.html (fetched 2012-06-01)

[Apfelmus; 2012] FRP - Upcoming API changes in reactive-banana 0.5, http://apfelmus.nfshost.com/blog/2012/01/01-frp-api-0-5.html (fetched 2012-06-01)

[Bengtsson, Bung, Gustafsson, Jeppsson; 2010] Haskell in javascript. Bachelor's thesis, Department of Computer Science and Engineering Chalmers University of Technology, Division of Computer Engineering Gothenburg University

[Björnesjö, Holm; 2011] JSHC - JavaScript Haskell Compiler, Bachelor's thesis, Department of Computer Science and Engineering, Gothenburg University

[Björnsson, Broberg; 2008] HJScript, http://hackage.haskell.org/package/HJScript (fetched 2012-06-01)

[Cherry; 2010] JavaScript Module Pattern In Depth, http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth (fetched 2012-05-31)

[Dijkstra; 2010] Haskell to JavaScript Backend, http://utrechthaskellcompiler.wordpress.com/2010/10/18/haskell-to-javascript-backend/ (fetched 2012-06-01)

[Ekblad; 2011] A language for functional web programming, Bachelor's thesis, Department of Computer Science and Engineering, Gothenburg University

[Ekblad; 2012] Haskell port of the *Glosie* application, https://github.com/valderman/glosie/tree/hsglosie (fetched 2012-06-06)

[Elliott; 2009] Push-pull functional reactive programming, in proceedings of Haskell Symposium

[Elliott, Hudak; 1997] Functional reactive animation, in proceedings of International Conference on Functional Programming

[GHC performance; Haskell Wiki] http://www.haskell.org/haskellwiki/Performance/GHC (fetched 2012-05-31)

[Golubovsky; 2007] YCR2JS, a Converter for YHC Core to JavaScript, http://www.haskell.org/haskellwiki/Yhc/Javascript (fetched 2012-06-01)

[Google Closure Compiler] https://developers.google.com/closure/compiler/ (fetched 2012-06-05)

[Marlow, Peyton Jones; 2004] How to make a fast curry: push/enter vs eval/apply, in proceedings of International Conference on Functional Programming

[Peyton Jones et al; 2002] The Haskell98 Report, http://www.haskell.org/onlinereport/ (fetched 2012-06-04)

[Hughes; 1984] Why Functional Programming Matters, http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html (fetched 2012-05-31)

[Mitchell; 2011] Yhc is dead, http://yhc06.blogspot.se/2011/04/yhc-is-dead.html (fetched 2012-06-01)

[Nazarov; et al] The GHCJS source repository, https://github.com/ghcjs/ghcjs (fetched 2012-05-31)

[Peyton Jones; 1992] Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine

[jQuery] http://jquery.com/ (fetched 2012-05-31)

[jQuery API documentation: map] http://api.jquery.com/jQuery.map/ (fetched 2012-05-31)

[Severance; 2012] Java Script: Designing a Language in 10 Days, Computer pp. 7-8, Feb. 2012

[Seipp; 2010] The Great Haskell Compiler Shootout, http://lhc-compiler.blogspot.se/2010/07/great-haskell-compiler-shootout.html (fetched 2012-06-01)

[Skårstedt; 2012] The Glosie source repository, https://github.com/nyson/glosie (fetched 2012-05-31)

[Stutterheim; 2012] Improving the UHC JavaScript Backend, http://www.norm2782.com/improving-uhc-js-report.pdf (fetched 2012-06-01)

[Tibell; 2011] Results from the state of haskell, 2010 survey. http://blog.johantibell.com/2011/08/results-from-state-of-haskell-2011.html (fetched 2012-06-01)

[Zakai; 2011] Emscripten: An LLVM-to-JavaScript Compiler, https://github.com/kripken/emscripten/blob/master/docs/paper.pdf (fetched 2012-06-01)