Thesis for the Degree of Licentiate of Philosophy

A Distributed Haskell for the Modern Web

Anton Ekblad

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology
and Göteborg University
Göteborg, Sweden 2015

A Distributed Haskell for the Modern Web
Anton Ekblad

Technical Report 145L
ISSN 1652-876X
Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology
and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Göteborg, Sweden 2015

Abstract

We present the design and implementation of a novel programming model and software development suite for interactive, distributed web applications using the Haskell programming language. The suite includes the JavaScript-targeting *Haste* Haskell compiler which improves on the current state of the art by producing smaller and leaner JavaScript code while preserving compatibility with standard Haskell as well as with the de facto standard GHC compiler.

We also describe the *Haste.Foreign* lightweight, portable interface for interoperating with JavaScript code, which allows boilerplate-free incorporation of JavaScript libraries in Haskell programs and vice versa. Haste.Foreign is implementable as a client library, and does not require any compiler modifications. While designed for the Haste compiler, the interface is portable across a range of Haskell dialects and high level target platforms.

Finally, we present the *Haste.App* programming model for distributed web applications, which abstracts over the separation of client and server to allow distributed applications to be written and type checked as a single program. The Haste.App model stands in stark contrast to the conventional way of developing web applications as separate client and server programs communicating explicitly over some network protocol. Haste.App reduces the amount of boilerplate code required to implement distributed web applications, and provides type safety across the network separating the client and server parts. This shortens development times and eliminates costly and embarrassing runtime failures at the boundaries between networked components. Like the foreign function interface, this programming model is implementable entirely as a library without any compiler modifications, and is thus similarly portable.

Contents

Contents **v**

Introduction **1**
1 Background 2
2 Related work 4
3 Conclusions 11
4 Future work 15

A Haskell compiler for browser environments **19**
1 Overview of the Haste compiler 19
2 The runtime system 22
3 Data representation 27
4 From STG to JavaScript 31
5 Optimizing JavaScript 39
6 Performance evaluation and discussion 44

Interoperating with JavaScript **51**
1 Background 51
2 An FFI for the modern web 55
3 Optimizing for safety and performance 60
4 Putting our interface to use 63
5 Performance 70
6 Limitations and discussion 73

Bridging the client-server gap **77**
1 Background 77
2 A seamless programming model 79
3 Implementation 86
4 Limitations and discussion 93

Bibliography **97**

Acknowledgements

I would like to thank my supervisors, Koen Claessen and Emil Axelsson, for their valuable support and guidance. Thanks for being a constant source of inspiration, enthusiasm and encouragement!

I would also like to thank my colleagues at the department for making this the greatest workplace I've known so far. Thanks for all the interesting discussions, for your valuable input on matters great and small, and for providing such a fun and stimulating working environment.

Finally, I would like to thank my family – my parents, my sister and my wonderful life partner Sofia – for all the love and support they have given me throughout the years. This thesis would not exist if not for you guys.

Introduction

We spend large parts of our daily lives in a web browser: everything from socializing with friends and dating, to shopping, filing our taxes and banking, increasingly takes place on the Internet, which to most people is synonymous with their web browser. In response to the public's ever increasing appetite for more, and more sophisticated, online services, JavaScript has emerged as the sole language of choice for client side web application development. The web as an application platform offers many benefits: it is a relatively uniform and standardized platform, web applications are easy to deploy, and web based interfaces offer a degree of commonality, lessening friction with new users. Any language community that wishes to stay relevant for developing end user-targeting applications would do well to keep a close eye on the development of the web application platform.

Layout of this thesis This thesis describes the design and implementation of a comprehensive programming environment for building web-targeting Haskell applications. Chapter 1 gives an overview of this programming environment, its background and related work, and the contributions made by this thesis. Chapter 2 describes the design and implementation of *Haste*, a Haskell compiler with a JavaScript backend. Based on the state of the art GHC Haskell compiler, Haste produces small, efficient JavaScript programs from idiomatic Haskell code. Chapter 3 describes the *Haste.Foreign* high level foreign function interface, which is specifically geared towards the challenges of interoperating with JavaScript code. This interface has a surprisingly lightweight implementation, and is portable across Haskell compilers and dialects as well as target languages. Chapter 4 describes the *Haste.App* web application programming model. Haste.App allows the development of client-server web applications which are type safe even across the network separating the client and server parts. This stands in contrast to traditional web applications, where the client and the server may be individually type correct, but not guaranteed to interoperate in a type safe manner.

Together with an extensive web standard library, these components make up the Haste development suite. The Haste suite has been used with good results in industry as well as in academia and in teaching, and is available as free software from its website at http://haste-lang.org.

While each chapter is followed by a short discussion and summary, the main discussion and analysis of the results of this thesis, including a discussion of related work and a summary of its contributions, are presented as a whole at the end of chapter 1.

1 Background

1.1 Unconventional compilation

The compilation of functional languages to unconventional architectures has a long and proud history. In the 1980's, functional languages performed poorly on stock architectures which led to the invention of specialized computer architectures such as the Normal Order Reduction MAchine (NORMA) [49]. Unconventional compilation targets were seen as a way to gain performance parity with the lower level languages of the time. This line of research went into decline as processors became faster and mass production of conventional chips made the economic case for special purpose architectures untenable, although the recent popularity of FPGAs seems to have the potential to reinvigorate the field [38].

Since the late 1990's, research into unconventional compilers has rather made a 180 degree turnabout: instead of producing blazing fast programs for newly invented physical machines, papers started appearing about producing relatively slow programs for established virtual machines, such as the Java Virtual Machine with its promised "write once, run anywhere" approach to program execution [4]. With the rise of the web as an application platform, just about every functional language now has a compiler targeting JavaScript: LuvvieScript [25] for Erlang, Ocsigen [3] for O'Caml, AFAX [41] for F#, several for Haskell [15, 19, 39], and so on.

1.2 JavaScript as a compilation target

As a target for compilation, JavaScript differs markedly from traditional assembly languages as well as more modern intermediate languages like the JVM or LLVM. Unlike the aforementioned target languages, JavaScript was designed for programmers, not compilers, and operates at a markedly higher level of abstraction. As a consequence, JavaScript has some unusual properties for a target language:

- The only numeric data type available is double precision IEEE754 floating point numbers. Despite this, said Number type still provides the full range of bitwise operators; all of which except one act as though their operands were 32 bit signed integers.

- There are no labels or unstructured jumps, even within delimited bodies of code. Any language features dependent upon arbitrary jumps must be simulated using higher level control structures.

- JavaScript has full support for first class functions, even allowing serialization and reification of arbitrary functions, which as discussed in chapter 3 has some positive implications for interfacing with JavaScript code.

JavaScript: the good parts One upshot of JavaScript's high level of abstraction is the possibility of a relatively direct mapping between source and target language. Source language functions are often representable by plain JavaScript functions, and expressive control structures make for relatively structured target code. Whereas significant tooling is required to productively debug programs compiled for native architectures, machine generated JavaScript code, and the compiler that produced it, is often debuggable to an extent by merely reading the generated code.

An execution environment meant for human consumption also brings to the table a rich set of APIs and runtime functionality. Unlike language implementors targeting more traditional environments, implementors targeting JavaScript can make use of the language's built-in facilities for garbage collection and other labour intensive parts of a language's runtime system instead of implementing their own. Both the language implementation itself and programs written in it are able to rely on the existence of a relatively rich set of standardized libraries for purposes ranging from string processing to graphical interfaces and network communication. This may in turn lead to less complex and labour intensive language implementations, smaller compiled programs, and greater cross platform compatibility, compared to languages targeting lower level platforms.

The other side of the coin JavaScript does have some significant drawbacks as a compilation target, however. As the language supports neither unstructured jumps nor native tail call optimization, supporting functional languages, which to a large extent require tail call optimization for correct operation, is problematic. In this way, the straightjacket imposed by structured control flow may hinder efficient compilation of some programs.

While the generous amount of functionality built into the language makes implementation of many features easier than in a lower level environment, the implementation enabled by said features is usually not ideal. Take as an example the memory behaviour of Haskell programs, which tend to allocate and discard large amounts of immutable data with a relatively short lifespan. This behaviour is not an ideal match for a JavaScript garbage collector which is tuned for programs which allocate relatively modest amounts of long-lived, frequently mutated data. While it is possible for implementors to use their own garbage collection schemes and fall back to a lower abstraction level, this generally entails giving up most or all of the benefits previously described [27].

The browser is also a constantly moving target. APIs are frequently introduced or modified, often with subtly differing semantics between different browsers, and new optimizations in JavaScript interpreters may render a program which used to be slow quite fast, or vice versa. While this is true to some extent for traditional architectures as well, they tend to move significantly slower and place a premium on backwards compatibility.

2 Related work

The Haste suite is not the only software development suite aiming to enable web development at a higher level of abstraction than what JavaScript natively provides. As previously stated, almost every popular language in existence has a JavaScript compiler these days. As discussing them all would be relatively impractical, this section focuses on the areas of research most immediately relevant to this thesis: compilers from functional languages to JavaScript and their foreign function interfaces, and web-targeting distributed programming environments.

2.1 Other compilers and FFIs

Clean The Clean compiler is able to generate JavaScript through the Sapl intermediate language [7]. Its compilation scheme is relatively similar to that of Haste, but the compiler uses a different source language as well as abstract machine and differs on certain key choices regarding data representation and runtime system. In particular, it uses an array model for the representation of algebraic data types as well as thunks, which we show in section 5 of chapter 2 to be relatively inefficient. Unfortunately, the actual compiler and benchmarks referred to in this paper are no longer available, making a direct performance comparison quite difficult.

The Clean language sports a foreign function interface which differs slightly from the rest of the compilers presented here. In Clean, the module

system makes a difference between *definition modules*, where abstract types and functions are declared, and *implementation modules*, where implementations are given for the types and functions declared in the corresponding definition modules. Instead of using a special "foreign import" syntactic form, Clean allows developers to write *system* implementation modules: modules where the implementations of functions defined in a definition module may be written in a language other than Clean [46]. However, only primitive types may be passed to this foreign code and no guarantees, making higher level interoperability cumbersome. Clean's FFI is thus more flexible than the foreign function interface of GHC, but less so than the solutions used by Fay, GHCJS, Haste or Idris.

Fay Haste's FFI was partially inspired by the foreign function interface of the Fay language, a "proper subset of Haskell that compiles to JavaScript" [16]. While the two are very similar in syntax, allowing users to import typed strings of host language code, Fay's solution is highly specialized. The compiler takes a heavy hand in the marshalling and import functionality, parsing the host language code and performing certain substitutions on it. While marshalling of arbitrary types is available, this marshalling is not easily controllable by the user, but follows a sensible but fixed format determined by the compiler. This approach makes sense, as the interface is designed to support the Fay language and compiler alone, but differs from our work which aims to create a more generally applicable interface.

GHCJS GHCJS [39] is, similar to Haste, a GHC-based compiler from Haskell to JavaScript, albeit with a different focus of development. Whereas Haste aims to produce small, fast JavaScript code and is willing to compromise certain features that are available in vanilla GHC to reach that goal, GHCJS aims to maximize compatibility with vanilla GHC at any cost. GHCJS compiles Haskell into continuation passing style, using a global trampoline to combat stack overflow, and partially manages its own heap on top of the JavaScript garbage collector. While this elegantly enables certain features which are not presently available in Haste – most prominently weak references – a heavy price has to be paid in terms of execution speed and code size as discussed in section 6 of chapter 2.

GHCJS provides no mechanisms of its own to facilitate the development of distributed web applications, but the *Haste.App* library described in chapter 4 could be implemented on top of GHCJS just as easily as on top of Haste.

GHCJS uses the relatively recent *JavaScriptFFI* GHC extension, which has unfortunately been rarely described outside a GHCJS context, to the point

of being conspicuously absent from even the GHC documentation. Much like Fay, this extension parses and performs substitutions over imported host language code to make imports slightly more flexible, allowing for importing arbitrary expressions rather than plain named functions. It also enables additional safety levels for foreign imports: *safe*, where bad input data is replaced by default values and foreign exceptions caught and marshalled into Haskell equivalents, and *interruptible*, which allows host language code to suspend execution indefinitely even though JavaScript is completely single threaded. This is accomplished by handing interruptible functions a continuation in addition to their usual arguments, to call with the foreign function's "return value" as its argument when it is time for the foreign function to return and let the Haskell program resume execution.

The JavaScriptFFI extension preserves the regular FFI's onerous restrictions on marshallable types however, and while GHCJS comes with convenience functions to convert between these more complex types and the simple ones allowed through the FFI, marshalling is not performed automatically and functions in particular are cumbersome to push between Haskell and JavaScript. Due to the complex runtime and memory management required for GHCJS to support weak references, Haskell functions can not be straightforwardly exported as library code, as this may cause memory leaks and other undesirable behaviors; exporting a GHCJS function essentially entails exporting a raw Haskell heap object.

Idris Idris is a dependently typed, Haskell-like language with backends for several host environments, JavaScript being one of them [6]. Like Haskell, Idris features monadic IO, but unlike Haskell, Idris' IO monad *is*, in a sense its foreign function interface. IO computations are constructed from primitive building blocks, imported using a function not unlike our *host* function described in section 2 of chapter 3, and parameterized over the target environment. This ensures that Idris code written specifically for a native environment is not accidentally called from code targeting JavaScript and vice versa.

Idris' import function does not necessarily accept strings of foreign language code, but is parameterized over the target environment just like the IO monad; for JavaScript-targeting code, foreign code happens to be specified as strings, but could conceivably consist of something more complex, such as an embedded domain-specific language for building Idris-typed host language functions.

UHC The UHC Haskell compiler comes with a JavaScript backend as well. Unlike Haste and GHCJS, UHC bears no relation to GHC, being

implemented from scratch using attribute grammars [15]. Like Fay, UHC provides automatic conversion of Haskell values to JavaScript objects, as well as importing arbitrary JavaScript expressions, with some parsing and wildcard expansion. Also like Fay, the JavaScript representation produced by this conversion is determined by the compiler, and is not user configurable. UHC does, however, provide several low level primitives for manipulating JavaScript objects from within Haskell, both destructively and in a purely functional manner.

2.2 Foreign interfaces with quasi quotes

Quasi quoting represents another, more radically different, approach to the problem of bridging with a host language [33]. Allowing for the inline inclusion of large snippets of foreign code with compile time parsing and type checking, quasi-quotes have a lot in common with our interface, even eclipsing it in power through anti-quotes, which allow the foreign code expressions to incorporate Haskell data provided that the proper marshalling has been implemented. Recent work by Manuel Chakravarty has extended the usefulness of quasi-quotes even further, automating large parts of the stub generation and marshalling required for using quasi-quoted host language code as a foreign function interface [9].

This usefulness comes at the price of a more involved implementation. Quasi quoting requires explicit compiler support in the form of compile time template meta programming as well as special extensions for running the quasi quoters themselves. In order to make full use of its compile time parsing and analysis capabilities an implementor also need to supply a parser for the quoted language.

2.3 Web-targeting distributed platforms

Several other approaches to seamless client-server interaction exist. In general, these proposed solutions tend to be of the "all or nothing" variety, introducing new languages or otherwise requiring custom full stack solutions. In contrast, Haste.App, our approach to writing distributed web applications, can be implemented entirely as a library and is portable to any pair of compilers supporting typed monadic programming. Moreover, Haste.App has a quite simple and controlled programming model with a clearly defined controller, which stands in contrast to most related work which embraces a more flexible but also more complex programming model.

The more notable approaches to the problem are discussed further in this section.

Conductance and Opa Conductance [24] is an application server built on StratifiedJS, a JavaScript language extension which adds a few niceties such as cooperative multitasking and more concise syntax for many common tasks. Conductance uses an RPC-based model for client-server communication, much like our own, but also adds the possibility for the server to independently transmit data back to the client through the use of shared variables or call back into the client by way of function objects received via RPC call, as well as the possibility for both client and server to seamlessly modify variables located on the opposite end of the network. Conductance is quite new and has no relevant publications. It is, however, used for several large scale web applications.

While Conductance gets rid of the callback-based programming model endemic to regular JavaScript, it still suffers from many of its usual drawbacks. In particular, the weak typing of JavaScript poses a problem in that the programmer is in no way reprimanded by her tools for using server APIs incorrectly or trying to transmit values which can not be sensibly serialized and deserialized, such as DOM nodes. Wrongly typed programs will thus crash, or even worse, gleefully keep running with erroneous state due to implicit type conversions, rather than give the programmer some advance warning that something is amiss.

We are also not completely convinced that the ability to implicitly pass data back and forth over the network is a unilaterally good thing; while this indeed provides the programmer some extra convenience, it also requires the programmer to exercise extra caution to avoid inadvertently sending large amounts of data over the network or leak sensitive information.

The Opa framework [47], another JavaScript framework, is an improvement over Conductance by introducing non-mandatory type checking to the JavaScript world. Its communication model is based on implicit information flows, allowing the server to read and update mutable state on the client and vice versa. While this is a quite flexible programming model, we believe that this uncontrolled, implicit information flow makes programs harder to follow, debug, secure and optimize.

Google Web Toolkit Google Web Toolkit [54], a Java compiler targeting the browser, provides its own solution to client-server interoperability as well. This solution is based on callbacks, forcing developers to write code in a cumbersome continuation passing style. It also suffers from excessive boilerplate code and an error prone configuration process. The programming model shares Haste.App's client centricity, relegating the server to serving client requests.

Cheerp Cheerp [45] is a C++ compiler targeting the web, written from the ground up to produce code for both client and server simultaneously. It utilizes the new attributes mechanism introduced in C++11 [52] to designate functions and data to live on either client or server side. Any calls to a function on the other side of the network and attempts to access remote data are implicit, requiring no extra annotations or scaffolding at the call site. Cheerp is still a highly experimental project, its first release being only a few months old, and has not been published in any academic venue.

Like Conductance, Cheerp suffers somewhat from its heritage: while the client side code is not memory-unsafe, as it is not possible to generate memory-unsafe JavaScript code, its server side counterpart unfortunately is. Our reservations expressed about how network communication in Cheerp can be initiated implicitly apply to Cheerp as well.

Sunroof In contrast to Conductance and Cheerp, Sunroof [5] is an embedded language. Implemented as a Haskell library, it allows the programmer to use Haskell to write code which is compiled to JavaScript and executed on the client. The language can best be described as having JavaScript semantics with Haskell's type system. Communication between client and server is accomplished through the use of "downlinks" and "uplinks", allowing for data to be sent to and from the client respectively.

Sunroof is completely type-safe, in the DSL itself as well as in the communication with the Haskell host. However, the fact that client and server must be written in two separate languages – any code used to generate JavaScript must be built solely from the primitives of the Sunroof language in order to be compilable into JavaScript, precluding use of general Haskell code – makes code reuse hard. As the JavaScript DSL is executed from a native Haskell host, Sunroof's programming model can be said to be somewhat server centric, but with quite some flexibility due to its back and forth communication model.

Ocsigen Ocsigen [3] enables the development of client-server web applications using O'Caml. Much like Opa, it accomplishes typed, seamless communication by exposing mutable variables across the network, giving it many of the same drawbacks and benefits. While Ocsigen is a full stack solution, denying the developer some flexibility in choosing their tools, it should be noted that said stack is rather comprehensive and well tested.

AFAX AFAX [41], an F#-based solution, takes an approach quite similar to ours, using monads to allow client and server side to coexist in the same program. Unfortunately, using F# as the base of such a solution raises the

issue of side effects. Since any expression in F# may be side effecting, it is quite possible with AFAX to perform a side effect on the client and then attempt to perform some action based on this side effect on the server. To cope with this, AFAX needs to introduce cumbersome extensions to the F# type system, making AFAX exclusive to Microsoft's F# compiler and operating system, whereas our solution is portable to any pair of Haskell compilers.

HOP, Links, Ur/Web and others In addition to solutions which work within existing languages, there are several languages specifically crafted targeting the web domain. These languages target not only the client and server tiers but the database tier as well, and incorporate several interesting new ideas such as more expressive type systems and inclusion of typed inline XML code [11, 13, 51]. As this thesis aims to bring typed, seamless communication into the existing Haskell ecosystem without language modifications, these languages solve a different set of problems.

Haxl Facebook's Haxl language, which specializes in fast data access, presents an interesting solution to the problem of concurrent access to distributed data sources [34]. A client side application must often fetch several distinct data sets in order to perform its functionality: a product view in a web shop may need to fetch information about the product, but also information about related products and about the user's shopping cart. These data sets are conceptually distinct and often not rendered in the same place, so in the interest of clarity we don't want to clump these data fetches together. Moreover, not all of this data may come from the same source, which further complicates any attempt at merging the requests. However, performing these fetches one after another is slow: each fetch needs to perform a complete network roundtrip, increasing the rendering time of the application by several times!

Marlow et al presents an abstraction for Haxl under which logically separate data fetches are merged and performed at the same time. Fetches from different data sources are performed in parallell and multiple fetches from the same source are batched, reducing the required latency of the set of fetches as a whole. The abstraction is based on an extension of Claessen's concurrency monad [12], but dispenses with explicit forking. Instead, the abstraction exploits the fact that every monad is also an applicative functor, handling batching, parallel requests and blocking implicitly in the <*> operator of the monad's *Applicative* instance.

3 Conclusions

This thesis presents a comprehensive programming environment for rich client web applications. This environment consists of the JavaScript-targeting *Haste* Haskell compiler, the *Haste.Foreign* high level foreign function interface, and the *Haste.App* library and distributed programming model for client-server web applications.

As a whole, the Haste development suite aims to simplify and advance modern web development through the application of core functional programming techniques in novel ways. While not yet as mature or polished as less exploratory programming environments, the Haste suite provides a compelling combination of distribution, type safety and ease of use. The Haste suite bridges the gap between the JavaScript and Haskell communities, allowing users to draw upon the best of both worlds; the user-centric design and the "bells and whistles" of contemporary web development, and the safety and correctness-focused, highly principled abstractions of the functional programming world.

The Haste compiler The Haste compiler provides a base upon which to build further research into client side web development using lazy, functional languages. It aims to provide fertile ground for experimentation and research as well as less exploratory web application development. To this end, Haste employs a minimalistic runtime system and a relatively straightforward translation scheme as described in chapter 2, and a sports an implementation that emphasizes code reuse and maximal utilization of previous work.

In addition, the Haste compiler improves upon the current state of the art in web-targeting Haskell dialects by providing superior performance and small code output while retaining near complete[1] compatibility with GHC Haskell. While these minor incompatibilities preclude or complicate certain use cases – higher order functional reactive programming being one – only the lack of weak references (unless supported by the underlying JavaScript execution environment) is intrinsic to the compiler's design. Considering the code footprint and performance benefits gained by this approach, as well as the ability to export Haskell code to native JavaScript programs without having to deal with complications in memory management, we deem this an acceptable sacrifice to make.

[1] Lack of support for weak references and the Template Haskell extension being the exceptions.

Haste.Foreign The Haste.Foreign interface enables frictionless interoperability between web-targeting Haskell code and its native JavaScript execution environment. Not only does it allow Haskell programs to draw upon the vast library of third party code in the JavaScript ecosystem, but it also allows the export of parts of Haskell programs as a JavaScript library, for the quite common case where a complete conversion of an existing JavaScript code base into Haskell is not an option. While designed and implemented for the Haste compiler, the concepts of the interface are applicable to other functional languages targeting a wide range of high level environments as well.

We have given a number of optimizations, improving the performance and safety of the interface and lightening the restrictions placed on the host environment. Additionally, we have given a library-only implementation of our interface for the Haste compiler, which is also portable across web-targeting Haskell dialects with a minimum of modification.

Finally, we have used this implementation to further extend our marshalling capabilities to cover functions, as well as generic default marshalling for arbitrary data types, contrasted our approach with a variety of existing foreign function interfaces, and demonstrated that our library does not introduce excessive performance overhead compared to the vanilla FFI.

While our interface is currently not applicable to Haskell implementations targeting low level, C-like environments, it brings significant reductions in boilerplate code and complexity for users needing to interface their Haskell programs with their corresponding host environment in the space where it *is* applicable: Haskell implementations for high level target platforms.

Haste.App With Haste.App we present a programming model which improves on the current state of the art in client-server web application development. In particular, Haste.App combines type safe communication between the client and the server with functional semantics, clear demarcation as to when data is transmitted and where a particular piece of code is executed, and the ability to effortlessly share code between the client and the server.

Our model is client-centric, in that the client drives the application while the server takes on the role of passively serving client requests, and is based on a simple blocking concurrency model rather than explicit continuations. It is well suited for use with a GUI programming style based on self-contained processes with local state, and requires no modification of existing tools or compilers, being implemented completely as a library.

As we saw in section 2, there exists a wealth of frameworks for dis-

tributed application development. While Haste.App is less versatile than some, for instance insisting on client centricity and a strict request-response model instead of allowing arbitrary communication or even cross-network access to mutable variables, we believe that this restriction is not so much a limitation as a strength. In nudging the programmer towards what we believe is a sensible approach to application development, we hope to reduce the complexity, and by extension the number of defects, in web applications written using Haste.App.

Similarly, in allowing only explicitly exported functions to be executed remotely, instead of allowing arbitrary client-side composition of server-side code, a significant source of security issues is eliminated. To illustrate why such arbitrary composition may be dangerous, imagine a situation where an application's server component supplies two operations: `fireMissiles`, which launches a flurry of ballistic missiles against some target, and `isPresident`, which checks whether the user is the president of the United States and thus authorized to launch missile strikes. Using these two operations, an application may implement the missile launch button as `onServer (when isPresident fireMissiles)`. If the server side computation were built on the client and then shipped to the server for execution, a malicious user could easily remove the `isPresident` check from the computation before shipping it off to the server for execution, allowing anyone with a web browser and basic web programming skills to launch missile strikes at will!

In order to make the same mistake with Haste.App, the programmer would need to export `isPresident` and `fireMissiles` separately, and explicitly call `onServer isPresident` followed by `onServer fireMissiles`. Not only does this explicit notation make the mistake much easier to spot, but it makes it much harder to make in the first place, considering that doing the right thing – exporting `when isPresident fireMissiles` as a single, atomic server side computation – is easier and requires less code than taking the insecure approach.

Contributions to the field In summary, this thesis makes the following contributions to the field:

- We present the *Haste* compiler which allows programs written in standard GHC Haskell to compiled to and executed performantly in a web browser environment, while allowing Haskell code to be incorporated in native JavaScript programs and vice versa. We describe in detail the compilation scheme and runtime system of the Haste compiler and analyze its performance, showing it to produce code which is significantly faster and smaller than the current state of the art.

- We investigate the performance impact of several optimizations relating to tail call elimination and trampolining, as well as several schemes for representing Haskell values in JavaScript, showing that the techniques used in the Haste compiler have a positive performance impact compared to these other techniques.

- We present a design for the the lightweight, compiler agnostic *Haste.Foreign* foreign function interface for high level target environments such as a web browser. Haste.Foreign is implementable in pure Haskell on top of the vanilla foreign function interface, without compiler modifications, and enables the import and export of complex, high level data such as algebraic data types and overloaded higher order functions without any boilerplate code. Programmers may override this automatic marshalling and take control over how data is marshalled using Haste.Foreign as needed.

- We give a reference implementation of the Haste.Foreign interface for the Haste compiler and evaluate its performance. We show this reference implementation to perform at least on par with the vanilla foreign function interface of GHC.

- We describe the novel *Haste.App* programming model for distributed web applications, using program slicing to allow applications to be written, type checked and compiled as a single program. Haste.App enables type safety across the network boundary, eliminating boilerplate code, bugs and incompatibilities in the communication between the client and server components. It also allows significant code sharing between the client and the server components.

- We give a reference implementation of Haste.App on top of the GHC and Haste compilers, showing that the programming model is implementable in pure Haskell, without compiler modifications.

- We survey the field of distributed web frameworks as well as foreign function interfaces for functional languages, and contrast the approaches taken in this thesis with the current crop of the field, arguing the efficiacy of our solution vis a vis these other frameworks.

Statement of contributions Chapter 1 presents the background and results of this thesis. It is part original work, part based on the two papers which chapters 3 and 4 are based on. The work for this chapter was performed in its entirety by the author. Chapter 2 is original work, a continuation of the author's Masters' thesis. Chapter 3 is a slightly revised

version of a paper currently under consideration for publication in the proceedings of the 2015 *Implementation and application of Functional Languages* workshop. The work – original ideas, design, implementation and writing – for chapters 2 and 3 was performed in its entirety by the author. Chapter 4 is a revised version of a paper published in the proceedings of the 2014 *Haskell Symposium* workshop, coauthored with Koen Claessen. The work on this paper – design of the programming model, implementation and writing – was mainly carried out by the author, while the idea to distribute applications by compiling the same program with two different compilers, as well as feedback and revision work on the paper itself, came from Koen Claessen.

4 Future work

Investigating alternative compilation targets JavaScript is the current de facto gold standard of web development, but it does have certain shortcomings as a compilation target as well as a major performance penalty compared to native code. Mozilla's *ASM.js* technology attempts to close this performance gap by compiling a low level subset of JavaScript into efficient native code before executing it [27]. With the recent announcement of *WebAssembly*, a cross-browser effort to create a common binary assembly language for web browsers, compilation for this type of target is starting to become an interesting proposition [17]. Investigating the viability of porting Haste's compilation scheme to ASM.js and WebAssembly opens up the possibility of writing even smaller, faster functional web applications, and may allow the application of traditional compilation techniques to a greater extent than when targeting JavaScript.

Improving the FFI While our interface is designed for web-targeting Haskell dialects, extending its applicability is generally a venue worthy of further exploration.

By combining two optimizations given in section 3 of chapter 3, the restriction of our *safe_host* function to only accept statically known strings and the elimination of calls to *eval* for statically known strings, it is possible to lift the restriction that a potential host language support dynamic code evaluation: if all foreign imports are statically known, and we are able to eliminate *eval* calls for all statically known functions, it follows that we are able to eliminate all *eval* calls. While the actual implementation of this idea has yet to be worked out, guaranteeing the complete absence of *eval* from the generated host code would remove the restriction that our host language supports dynamic code evaluation at runtime, notably

making our interface implementable on recent versions of the Java Virtual Machine. Implementing this interface for the Java Virtual Machine, with the prerequisite Haskell-to-JVM compiler, would lend additional applicability to our interface.

Haste.Foreign does not currently catch and marshal host language exceptions, but requires foreign language code to take care of any exceptions. While the actual implementation is quite specific to a particular host environment, automatically converting exceptions would be a useful feature even so. Investigating the degree to which this feature could be implemented in a host platform agnostic manner would be a possible extension of this work.

Due to the hard requirement that our host language be garbage collected, our interface is not currently applicable in a C context. This is unfortunate, as C-based host environments are still by far the most common for Haskell programs. It may thus be worthwhile to investigate the compromises needed to lift the garbage collection requirement from potential host environments.

Information flow control Web applications often make use of a wide range of third party code for user tracking, advertising, collection of statistics and a wide range of other tasks. Any piece of code executing in the context of a particular web session may not only interact with any other piece of code executing in the same context, but may also perform basically limitless communication with third parties and may thus, inadvertently or not, leak information about the application state. This is of course highly undesirable for many applications, which is why there is ongoing work in controlling the information flow within web applications [26].

While this does indeed provide an effective defense towards attackers and programming mistakes alike, there is value in being able to tell the two apart, as well as in catching policy violations resulting from programming mistakes as early as possible. An interesting venue of research would be to investigate whether we can take advantage of our strong typing to generate security policies for such an information flow control scheme, as well as ensure that this policy is not violated at compile time. This could shorten development cycles as well as give a reasonable level of confidence that any run time policy violation is indeed an attempted attack.

Generalized distributed computing While the two most prominent parts of a web application are the client and the server-side program, they are not the only pieces of the puzzle by far. Modern web applications often make use of several external services, and the server component is often

connected to at least one database as well. It is also not hard to imagine an application making use of more specialized code on the client as well as on the server, possibly adding components written in several different domain-specific languages to the mix as well.

In order to safely and efficiently program such heterogeneous systems, the current Haste.App model is not enough. A promising line of future research investigates possible ways of generalizing this programming model to enable clients to utilize a range of different, possibly chained, servers. The performance of this generalized programming model might be enhanced by incorporating the work of Marlow et al on concurrent data access [34].

High performance JavaScript through EDSLs The code produced by Haste is generally relatively performant compared to that produced by other JavaScript-targeting compilers for lazy functional languages. However, there is still quite a wide performance gap between lazy functional languages and performance oriented languages like C and C++, even without factoring in the overhead of being executed on a JavaScript virtual machine. Performance-oriented embedded domain specific languages such as Feldspar [2] have been used with good results to turn high level functional code with some restrictions into highly optimized C programs. By modifying Feldspar to produce JavaScript code, it may be possible to produce code which is several times faster than what can be produced from a full Haskell program, especially if targeting the high performance ASM.js [27] subset of JavaScript.

Moreover, by using the techniques described in chapter 3, it should be possible to create a seamless bridge between Haskell and Feldspar programs, automatically compiling and loading Feldspar code on demand. This would allow a developer to implement performance intensive calculations in the fast but restricted Feldspar language while still using standard Haskell for high level business logic, completely transparently and without losing type safety.

Real world applications As the Haste suite is relatively new technology, it has yet to be used in the creation of large scale applications. While we have used it to implement some small applications, such as a spaced repetition vocabulary learning program, a cloud-based media player and a more featureful variant on the chatbox example given in chapter 3, further investigation of its suitability for larger real world applications through the development of several larger scale examples is an important area of future work.

A Haskell compiler for browser environments

This chapter describes the design and implementation of *Haste*, a web-targeting Haskell compiler. Section 1 gives a general background to the problem and a brief introduction to the overall design. Sections 2 and 3 describe the implementation of the compiler's runtime system and the in-memory representations of the constructs used by the Haskell language and the runtime itself. Sections 4 and 5 describe in detail the actual translation from Haskell to JavaScript and the program transformations and intermediate formats used throughout the process. Finally, section 5 evaluates the performance of the Haste compiler relative to the current state of the art, investigates the performance impact of the optimizations, runtime and data representations chosen, and summarizes our contributions to the field.

1 Overview of the Haste compiler

Countless contributors have spent thousands upon thousands of person hours developing efficient Haskell compilers, as well as improving the JavaScript ecosystem. Duplicating this effort by attempting to create from scratch a highly optimizing compiler from Haskell to JavaScript, complete with custom garbage collection, JavaScript optimization and runtime facilities, would demand quite significant effort – well beyond the scope of this thesis. Instead, the Haste compiler adopts, somewhat tongue in cheek, a single guiding principle: *it's someone else's problem*.

Background Considering this principle, it may seem odd to implement a Haskell to JavaScript compiler at all. After all, both the GHCJS [39] and UHC [15] Haskell compilers are perfectly able to produce JavaScript executables. Unfortunately, both the aforementioned compilers tend toward very large code output. While the size of a binary may not be very

interesting when it is mainly stored on and executed from a hard drive with several terabytes of storage capacity, it matters quite a bit when delivered repeatedly to clients over an expensive and possibly slow network connection.

The UHC compiler also has the problem of producing relatively slow programs. A program compiled with UHC is often more than an order of magnitude slower than the same program compiler with GHC [19], making it a poor starting point for a JavaScript backend which will invariably have a performance handicap relative to any native compiler. UHC also lacks support for many popular Haskell extensions supported by the GHC compiler, making it a relatively poor starting point for a client side Haskell web development suite.

GHCJS, on the other hand, is today a more promising starting point. Based on GHC, it boasts excellent compatibility with language features and third party code, and produces relatively fast programs to boot. However, at the time work started on Haste, GHCJS was in its infancy, still far from being usable in a larger context. Since then, GHCJS has improved by leaps and bounds, but still retains the aforementioned propensity towards large code output. Additionally, GHCJS aims to implement GHC Haskell as closely as possible, even where it is not obvious that doing so makes sense in a browser context.

Haste aims to produce significantly smaller output than either GHCJS or UHC. It also explores a different point in the design space than GHCJS: instead of emulating vanilla GHC to the extent possible, it aims to do so to the extent possible *while producing reasonably small, fast and idiomatic JavaScript*.

1.1 Outsourcing to GHC

There is really only one freely available industrial strength Haskell compiler: GHC. The bulk of the work that goes into improving the Haskell language happens in GHC, including a wide range of non-standard language extensions which have by now become so ubiquitous that it is hard to imagine writing Haskell code without them. To avoid having to continuously chase after GHC, never quite keeping up on performance or features, Haste instead incorporates it to provide a major part of the compilation process. Figure 1 provides an overview of the Haste compiler, and how it integrates with GHC.

However, while deciding to outsource compilation to GHC is all well and good, this is not the only decision that needs to be made. A Haskell program evolves through several intermediate formats during GHC's compilation process. When targeting a high level language such as JavaScript, which of

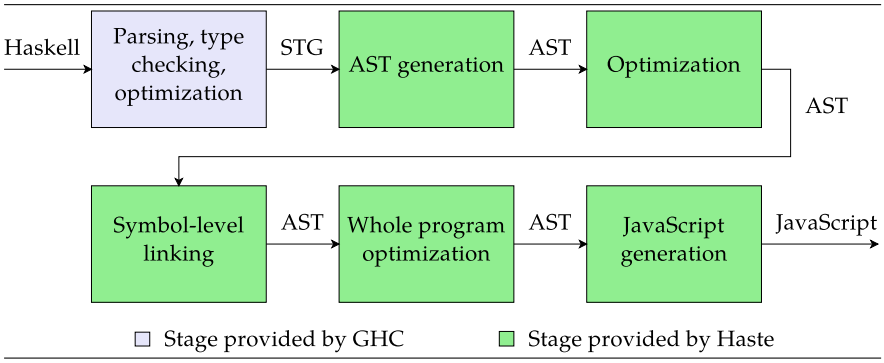

Figure 1: The Haste compilation pipeline.

these intermediate languages makes the most sense to use as our source language?

- Plain *Haskell source* comes in several flavors within GHC: parsed, type checked, and desugared, where syntactic sugar such as *do*-notation has been translated into equivalent unsugared Haskell expressions. The parsed source is first type checked and the type checked source is then desugared. A shared characteristic of all these flavors is that no optimizations have been performed at this stage.
- The desugared Haskell source is then translated into GHC's *Core* intermediate language, more formally known as *System FC*. System FC is essentially a non-strict, higher order polymorphic lambda calculus extended with algebraic data types. This is the stage where GHC performs most of its optimizations.
- The optimized Core is translated into the *STG* language. The main difference between STG and Core is that in STG the evaluation of thunks is explicit whereas in Core it is implicit. Further optimizations are performed at this stage.
- Finally, the optimized STG is translated into *C--*, a high level assembly variant, in preparation for code generation. This language is geared towards native code generation, and contains information relevant to register allocation, garbage collection and other pertinent low level details.

At first glance, as our purpose is code generation, using C-- as our input language would seem to make sense, as this is the role it performs within

GHC compilation pipeline. Certainly, we want to plug into the pipeline at as late a stage as possible in order get the most out of the work put into the GHC compiler. However, as described in chapter 1, JavaScript has several idiosyncrasies when compared to traditional low level compilation targets. As we want to make use of JavaScript's high level features and avoid low level concepts which are not easily or efficiently representable in JavaScript – pointers and arbitrary jumps being chief among them – C-- is too low level a target.

Instead, the next step up – STG – provides a sweet spot: memory management and function representation is still implicit, while most of the optimizations implemented by GHC have been applied at this stage and thunk evaluation has been made explicit.

2 The runtime system

A language implementation is not only defined by its translation into executable code and its chosen data representations, but also by its *runtime system*: the supporting functions and processes that a compiled program relies on for correct operation. While the GHC compiler comes with a complete, highly tuned runtime, it is much too geared towards execution on actual machine architectures for our purposes. Ideally we would like to make use of the browser's JavaScript runtime as much as possible, to avoid reinventing any wheels and to reduce the size of the code that will eventually be shipped to users' browsers.

Memory management, function definition and application, and basic program execution can trivially be outsourced to the JavaScript engine. There are however three major concepts which have no counterpart in JavaScript, and which we consequently need to implement ourselves: laziness, tail call elimination, and curried functions.

2.1 Laziness

The concept of laziness – also known as *lazy evaluation* or *call by need semantics* – refers to the Haskell property that no piece of data is evaluated unless it is needed and that the result of said evaluation is shared, preventing programs from performing unnecessary computations. This property is sometimes a great boon for both clarity and modularity, as described in [28]. Unfortunately, this boon comes at the price of reduced performance. This is particularly troublesome when the target language is a high level language with no inherent support for laziness, ruling out pointer tagging and other low level optimizations used by vanilla GHC [37].

Laziness is customarily implemented using *thunks*, updatable heap objects which either point to a concrete value, or to a block of code computing that value – the *body* of the thunk [35]. When a thunk is evaluated for the first time, its body is invoked to compute its value, which is then cached to avoid recomputation. Evaluating the thunk again merely returns the cached value computed during that first evaluation. There also exist a class of non-updatable thunks, which are guaranteed to be evaluated at most once and which thus do not need to cache the result of their computation.

In Haste, thunks are implemented using a JavaScript class T, which has two fields: f, which initially points to a closure containing the body of the thunk; and x, which initially contains a reference to a sentinel object specifying the thunk's behavior on evaluation: whether the thunk is updatable or not. After evaluation, this field will contain the thunk's computed value. To reduce pressure on the garbage collector and to avoid unnecessary computation, Haste aggressively optimizes away thunks which are statically known not to compute $\perp$ and to perform no computation beyond allocation – literal constants and numeric types being two common instances – which often leads to such thunks being represented solely by their raw values.

When a thunk t is evaluated the Haste runtime first determines whether t is an instance of T, and immediately returns t unchanged if it is not, as we can then deduce that this is one of the "raw value thunks" mentioned above. After the runtime has established that t is indeed a "proper" thunk, $t.x$ is inspected to determine how it should be handled. If $t.x$ is a reference to the *__updatable* sentinel object, the thunk is updatable.

In this case, we overwrite $t.f$ with another *__blackhole* sentinel object, indicating that the thunk is being evaluated. *Blackholing*, as this is commonly called, has two purposes. First, it indicates to the garbage collector that the function computing the thunk is no longer reachable, so that the thunk does not prevent collection of any heap objects mentioned in its body. Second, it allows for the detection of infinite loops: if we try to evaluate a thunk which has already been blackholed but not yet completely evaluated, we know that we have an infinite loop and can throw an appropriate exception [35].

After blackholing, we call $t.f$, the body of the thunk, in order to compute its value. When the body returns, $t.x$ is updated with the computed value, which is then finally returned to the calling function as the resulting expression of evaluating the thunk.

If $t.f$ is instead a reference to the *__blackhole* object and $t.x$ still refers to *__updatable*, we know that we have entered an infinite loop and terminate the program using a host language exception. This does not detect infinite

recursion in non-updatable thunks, but considering that non-updatable thunks are so by virtue of only ever being entered once, infinite recursion in a non-updatable thunk is impossible. If $t.x$ is *not* a reference to *__updatable* at this point, we know that t has already been evaluated, and so we simply return $t.x$.

2.2 Proper tail recursion

In functional programs, *proper tail recursion*, or *tail call optimization* as it is sometimes called, is of utmost importance. In the absence of loops, programmers must be able to use recursion to arbitrarily repeat tasks, possibly infinitely many times. However, the conventional implementation of function calls involves creating a new stack frame for each call, leading to infinite (or just deeply nested) recursive programs eventually running out of memory.

Fortunately, a special class of function calls can easily be implemented in constant space. A *tail call*, as these functions calls are called, is a function call which occurs in *tail position*: when the call is the last thing that happens before the calling function returns. Whenever you see a function definition reading `f x = g (... x ...)`, you are looking at a tail call.

A traditional compiler can exploit the fact that nothing can happen between a tail call and the return of the calling function. Since this means the entire stack frame used by the caller is not used after the tail call, it is possible to reuse this stack frame for the tail call by appropriately arranging the arguments to the tail callee in the current stack frame and replacing the function call with a jump instruction. This process is what is referred to as tail call optimization.

Trampolining Unfortunately, when compiling to a high level language, we do not have the luxury of arbitrary jumps at our disposal, making us dependent on the target language natively supporting tail call optimization. The quite recently ratified sixth edition of the *ECMA-262* standard introduces native support for tail call optimization to the JavaScript language but it will be quite some time before this support is ubiquitous in JavaScript engines, and even longer until it can be relied upon to be present in most users' devices [55].

Until then, we must simulate tail calls using a construction known as a *trampoline*: whenever a function wishes to make a tail call, it does not make the call itself but rather returns a closure performing the call. Each call site is then instrumented with a loop which inspects the call's return value. If a tail call closure is returned, the closure is called and the loop performs another iteration, repeating the process. If a value which is not a

tail call is returned, we know that the chain of tail calls has ended for now and return the value as we normally would [53]. The name of the construct lends a helping hand in gaining an intuitive understanding of the principle: functions "bounce" on the trampoline for as long as they keep tail calling, and "jump off" as soon as they are done.

While the performance impact of trampolining on the Haste compiler has not been explicitly measured, Baker gives the performance impact as a 2-3 times slowdown for a C implementation. With Haste's implementation of trampolining, each tail call causes two additional function calls: one to the trampolining instrumentation, which is not manually inlined to avoid complicating the code generator, and one to the closure making the tail call itself. Considering this, along with the fact that tail calls are always made to functions which are not statically known under this scheme, it is reasonable to assume at least a similar slowdown for Haste.

Through various optimizations, it is possible to reduce the performance impact of trampolining. The measures taken by Haste to this end are described in section 5. The significant performance boost obtained when optimizing and, where possible, removing the trampolining instrumentation lends some credibility to the above assumption that trampolining takes a relatively heavy toll on performance.

2.3 Curried functions

In Haskell, there is conceptually no such thing as a function of multiple arguments: all functions have the type $a \to b$. This does not mean that it is impossible to write Haskell functions over multiple arguments – a quite unreasonable proposition. Instead, a binary function $f : (a, b) \to c$ is encoded as a unary function $f : a \to (b \to c)$ returning a second function, which captures the argument of the first to form a closure, with n-ary functions following the same pattern. Such functions are known as *curried functions*. This scheme makes partial application of functions quite natural as there is simply no conceptual difference between applying, for instance, a ternary function $f : a \to b \to c \to d$ to one, two or three arguments; the only difference is in the return type being $b \to c \to d$, $c \to d$ or d respectively.

n-ary functions While convenient for the user, the efficient implementation of curried functions can be tricky. Haste's implementation quite closely follows the one given by [36]. Implementing the application of an n-ary function to m arguments as a series of m unary function applications would be quite inefficient, so n-ary Haskell functions are represented in generated code as n-ary JavaScript functions. This brings us to another

Name	Purpose
fun	Reference to the function under application.
args	Array of arguments to which *fun* has been partially applied.
arity	The arity of the PAP; equivalent to *fun.length* − *args.length*.

Table 2.1: Structure of a partial application object

conundrum: when all functions are conceptually unary, what does the term "n-ary function" even mean?

For performance reasons, given a function $f : a \to b \to c$ we want to represent f as a function from two arguments a and b to a first order value c to the extent possible, as opposed to a function from a to another function $b \to c$. GHC's optimizer already does a good job determining where this is possible, so an STG lambda expression $\lambda\ x_1\ ...\ x_n \to ...$ of syntactical arity n is always represented by Haste as the corresponding JavaScript function $function(x_1,\ ...\ x_n)\ \{...\}$ of the same arity. Whenever the arity of a function can be statically determined, we can use JavaScript's native function calls to apply it to its arguments.

From here on, we will use the term "n-ary function" to mean a function that has an STG representation with syntactical arity n.

Partial applications However, due to partial application it is not always possible to determine the representation arity of a function, as described by Marlow and Peyton Jones [36]. Thus, partial application gives rise to another function representation: *partial application objects*, or *PAPs* for short. A PAP simply consists of a reference to a non-PAP function object, a list of arguments to which the PAP has been applied so far, and the arity of the PAP object. The arity of the PAP object is always strictly less than or equal to the arity of the underlying function applied to zero arguments. The members of a PAP are listed in table 2.1. Note that in JavaScript the *length* property is, slightly confusingly, used to indicate the arity of a function, as well as the length of an array.

Generalized function application When encountering a function whose arity cannot be statically determined, the code generator can no longer punt the function application to JavaScript's native facilities. It must instead generate a call to an *apply* runtime function which performs a far more

heavyweight function application, inspecting the function object f being applied and and an array of given arguments *args*:

- If f is not a PAP, let f be a new PAP using f as the function reference, the empty list as its list of arguments, and f*.length* as its arity.
- If *f.arity* $=$ *args.length*, the application is no longer partial and *f.fun* can be immediately applied to *f.args*++*args* through JavaScript's native function call.
- If *f.arity* $>$ *args.length*, the function is being partially applied. In this case, *apply* creates a new PAP from *f.fun* and *f.args*++*args* with its arity set to *f.arity* $-$ *args.length*.
- If *f.arity* $<$ *args.length*, the function is being overapplied, indicating that f returns a function which expects to receive the extra arguments from *args*. In this case, *f.fun* is applied to the first *f.arity* elements of *args* appended to *f.args*. Its return value is then applied to the remaining elements of *args* using *apply*.

Additionally, due to the quite heavy nature of this function application procedure Haste's runtime provides specialized versions for functions of several common aritites, making the actual use of the general application relatively rare.

3 Data representation

Closely related to the implementation of the runtime system, is Haste's encoding of Haskell values into their JavaScript counterparts. The kinds of data operated upon by programs written in JavaScript and Haskell respectively are generally quite different. While both languages share the concept of first class functions, Haskell also provides the traditional set of types for compiled languages – signed and unsigned integers of various bit widths as well as single and double precision floating point numbers – as well as the algebraic data types common to most functional languages. JavaScript, on the other hand, only provides a single numeric type – sometimes behaving as a double precision floating point number, sometimes as a 32 bit integer – plus strings, arrays and objects consisting of key–value mappings. With this impedance mismatch, finding the appropriate mapping from Haskell values to their JavaScript equivalents is not always a straightforward proposition, even without the additional complication of laziness which needs to be handled as performantly as possible. Table 2.2 gives an overview of how different Haskell types are translated into JavaScript.

Haskell type	JavaScript representation
`Char`	`Number`
`IntN/WordN` ($N \leqslant 32$)	`Number`
`IntN#/WordN#` ($N \leqslant 32$)	`Number`
`Int64/Word64`	`goog.math.Long`
`Integer`	`goog.math.Integer`
Algebraic data types	`Object`
Functions and closures	`Function/Object`

Table 2.2: Haste type representations

3.1 Numeric types

As JavaScript only has a single numeric type, the choice of representation for machine types is severely restricted: if we want any semblance of performance, we are forced to rely on the *Number* type. As the internal representation of Number is – at least nominally – a double precision IEEE754 floating point number we are able to represent any floating point value, or any integer value of 53 bits or less due to the 53 bit mantissa of double precision floating point numbers. For our purposes this essentially makes the browser a 32 bit architecture, and all integer arithmetic in Haste is consequently performed modulo 2^{32}.

Note that, as described in section 3.2, certain lifted types are actually represented in JavaScript as though they were unlifted. This crucially includes numeric types which is why table 2.2, perhaps surprisingly, gives the same representation for lifted and unlifted types.

Machine integers Using a floating point representation for integer types may seem like an exceptionally bad idea. Not only is floating point math generally slower than its integer counterpart, but it also lacks a wide range of operations expected of integers: modular arithmetic, integer division, bitwise operations, etc. Fortunately, JavaScript recognizes the need for these operations. Somewhat bizarrely, values of the Number type provide a modulo operator, a 32 bit integer multiplication operation, and the normal set of bitwise operations, under which values of the type behave as 32 bit

signed integers.[1]

As Number values behave as floating point numbers under all other arithmetic operations, we must manually ensure that the results of those operations never exceed the bounds set by the machine types we are using them to emulate. Thus, arithmetic operations on machine integer types are performed modulo 2^n by taking the bitwise *OR* of the result and zero, which results in a number in the 32 bit signed integer range as described above. This unfortunately comes with a performance hit compared to the judicious insertion of range-preservation operations by a human programmer; although JavaScript engines try hard to identify integer arithmetic and eliminate any intermediate conversions, this analysis is only an approximation.

Integers and 64 bit arithmetic While the Number type is large enough to accommodate all machine types, provided that we treat the browser as a 32 bit machine, it does unfortunately not suffice when it comes to representing 64 bit integers or the arbitrary precision *Integer* type, both of which are frequently used in Haskell code.

For these types, we must instead come up with an appropriate representation of our own. Implementing efficient arbitrary precision integer arithmetic from scratch would take significant time and effort – as evidenced by the use of the *GMP* numeric library for this purpose by vanilla GHC – and while 64 bit arithmetic is not quite as daunting, an efficient implementation is still a decidedly non-trivial proposition. In the spirit of *someone else's problem*, Haste instead uses the *long.js* [56] and *bn.js* [29] libraries to represent 64 bit integers and arbitrary precision integers respectively.

3.2 Algebraic data types

Algebraic data types are the most obvious example of a Haskell concept which does not map directly onto JavaScript. Hence, we must make do with what facilities it does give us to model compound types: arrays and objects. Haste uses several different representations for algebraic data types, depending on the characteristics of the particular type.

- For *enumeration types* – types with one or more data constructors, each of which has exactly 0 arguments – Haste uses a Number identifying the data constructor used to construct a value of the type as its representation. The *Bool* type is a special case of this, where the *True*

[1] Except for the *zero-fill right shift* operator, which treats its left operand as a 32 bit *unsigned* integer.

and *False* constructors are represented by their JavaScript `true` and `false` constants respectively.

- *Newtype-like types* – types with exactly one constructor, with a single, unlifted argument – are represented by the argument itself. Due to the introspection-based tagging approach to representing thunks described in section 2.1, a thunk may be represented solely by its result, removing the need to wrap unlifted types in lifted ones at runtime. Removing this layer of indirection provides a considerable reduction in the amount of packing and unpacking code needed for numeric calculations, as well as reduced memory usage.

- All other algebraic data types are represented as JavaScript objects with a member id_{tag} identifying the tag of the data constructor used to create the value, and the constructor's arguments making up the rest of the object's members $id_{1..n}$.

As JavaScript engines are in constant flux, choosing an overall "best" in-memory representation for algebraic data types is not entirely straightforward. The memory consumption, allocation time, lookup time and other properties of JavaScript arrays relative to those of JavaScript objects vary significantly from JavaScript engine to JavaScript engine and even from version to version. As such, this is still an active area of experimentation, with the "optimal" representation of algebraic data types still a matter of debate.

For Haste's use case, the main contenders are plain arrays, anonymous objects, and – for lack of a better word – "classy" objects: objects created using the *new* keyword and a constructor function. Arrays and anonymous objects share the characteristics of being simple to create: both can be constructed using simple literals, giving them a small footprint when it comes to code size and making them trivial to produce for the code generator. Classy objects, on the other hand, have one constructor function per JavaScript type, potentially giving the JavaScript engine valuable information regarding the structure of such values which may be used to guide optimizations. It also makes the syntax of their creation slightly wordier, and requires some boilerplate code for each type present in a program.

Haste employs classy objects for its representation of algebraic data types. The aforementioned advantages of providing the JavaScript interpreter with more static information seems to pay off greatly for most programs, making classy objects significantly faster than its two competitors. The performance of the three different methodologies is evaluated in section 6.3.

Haskell expression	JavaScript representation
`Just x :: Maybe t`	`{_: 1, a: x}`
`Nothing :: Maybe t`	`{_: 0}`
`(x, y, z) :: (t0, t1, t2)`	`{_: 0, a: x, b: y, c: z}`
`(x : y : []) :: [t]`	`{_: 1, a: x, b: {_: 1, a: y, b: {_: 0}}}`
`LT :: Ordering`	`0`
`I# 42# :: Int`	`42`

Table 2.3: ADT representations in JavaScript

Table 2.3 shows the JavaScript representations of a selection of different expressions of algebraic data types.

4 From STG to JavaScript

As mentioned in section 1, Haste uses GHC's STG intermediate representation as the input language of its JavaScript translation. In this section we take a cursory look at the STG as an intermediate language, Haste's intermediate JavaScript representation, and how the two combine to produce, in the end, an executable JavaScript program from Haskell source code.

4.1 The STG language

STG is more than an intermediate representation, however: it is a complete abstract machine, with well defined data representations and compilation strategies [42]. As STG is intended for compilers targeting a low level, native platform, it is not always possible or desirable for Haste to adhere to these representations. If it were, this entire chapter would be quite unnecessary except as a slightly updated recap of the cited paper! From here on, we use the term "STG" to refer to the STG language, giving our own translation into JavaScript to set it apart from the original abstract machine itself.

The syntactic forms of the STG language, slightly simplified for readability, are given by table 2.4. A valid STG program must adhere to the following rules:

Construction	Purpose
l	Machine value literal
v	STG name
$f\ a_{1..n}$	Function application
$conApp(c,\ a_{1..n})$	Data constructor application
$opApp(op,\ a_{1..n})$	Primitive operation application
$\lambda v_{1..n}.e$	Lambda abstraction
$case\ v_0@e_0\ of$ $c_1\ v_{1..m}\ \rightarrow\ e_1$ ... $c_n\ v_{1..m}\ \rightarrow\ e_n$	Conditional expression
$let\ v = e_0\ in\ e_1$	Let-binding
$let\_no\_escape\ v = e_0\ in\ e_1$	Let-no-escape-binding

Table 2.4: The STG language

- Only atoms – identifiers and literals – are permitted as arguments to functions, data constructors or primitive operations.
- Application of data constructors and primitive operations is always saturated.
- Nullary functions are permitted, and are interpreted as thunks.

The first property is of little consequence to our code generation apart from making our compilation scheme slightly simpler. As we are generating code for a high level language, we don't care much about whether function arguments are identifiers or other expressions and a post generation optimization pass removes most of the extra identifiers introduced by this property by replacing redundant assignments with simple expression substitution. The other two rules have practical implications however, as we shall see when we discuss Haste's translation from STG to JavaScript.

No escape? The *let_no_escape* construct may require some additional explanation. Like the normal let-binding, a let-no-escape binding binds a

variable to an expression. Unlike a normal let-binding, however, it has some rather severe restrictions. A let-no-escape binder is guaranteed to never escape the scope of the bind – for instance by being referenced in a thunk that escapes the bind – and to always be tail called at least once before the stack is unwound. Thus, it can be thought of as encoding a *join point*; a labeled block of code which may be entered using a simple jump instruction and does not need to be stored on the heap. This is indeed its use within the GHC compiler. Its treatment by Haste is discussed further in section 4.3.

Primitive operations In addition to the functionality expected of a language based on the lambda calculus, STG also supports *primitive operations*: the smallest building blocks that make up any program, defined outside the language itself. These operations are made up of functionality spanning everything from logic and arithmetic over various machine types, to primitives implementing mutable arrays, concurrency and CPU-specific vector instructions. As there are several hundred different primitive operations, giving their translations in this thesis would be rather impractical. The nature of their being basic building blocks also makes the vast majority of them rather uninteresting. Thus, we are content to note that Haste implements all primitive operations required to support its feature set. Efficient JavaScript implementations of some primitive operations, mainly those relating to pointer arithmetic, is still an area of active work, however.

4.2 The simplified JavaScript AST

Haste does not immediately translate STG into JavaScript program text but into an intermediate format, slightly uncreatively referred to simply as "the AST". The AST is in essence a slightly restricted abstract representation of JavaScript, with the added concepts of thunks, evaluation and tail calls. Like JavaScript, the AST is divided into statements and expressions. Tables 2.6 and 2.5 show the expressions and statements of the AST respectively.

Statements A program is, at the top level, a series of assignments terminated by a *stop*, with one of the assignments being the program's entry point function. Statements are divided into two groups: terminating and non-terminating statements. The former group is made up of the different ways in which an AST subprogram may end. The *continue* and *forever* constructs correspond directly to the JavaScript statements `continue` and `while(true) { ... }` respectively. It should not come as a surprise that *continue* is a terminating statement as its JavaScript semantics simply transfer control back to the top of the innermost enclosing loop. Less obvious is the case

Statement	Purpose
$switch(exp)\ \{\ c_1 :\ stm_1,\ ...,\ c_n :\ stm_n\ \}\ ;\ stm_{cont}$	Conditional statement
$(exp_0 \parallel var\ v) := exp_1\ ;\ stm_{cont}$	Assignment
$forever\ \{\ stm\ \}$	Infinite loop
$continue$	Jump to head of loop
$stop$	No-op break
$ret_f\ exp$	Return from function
$ret_t\ exp$	Return from thunk
$tailcall\ f(x_1,\ ...,\ x_n)$	Tail call invocation

Table 2.5: Statements in Haste's AST

Expression	Purpose
v	A variable
l	A literal JavaScript value
$exp_1\ (+\parallel -\parallel *\parallel ...)\ exp_2$	Binary operator expression
$fun(v_1,\ ...,\ v_n)\ \{\ stm\ \}$	Function creation
$f(exp_1,\ ...,\ exp_n)$	Function call
$f_{tr}(exp_1,\ ...,\ exp_n)$	Trampolined function call
$[exp_1,\ ...,\ exp_n]$	Array creation
$exp_{arr}[exp_{ix}]$	Array indexing
$\{id_{tag} :\ l,\ id_1 :\ exp_1,\ ...,\ id_n :\ exp_n\}$	Object creation
$exp.id$	Object member lookup
$thunk(fun()\{\ stm\ \}, upd)$	Thunk creation
$eval(exp)$	Thunk evaluation

Table 2.6: Expressions in Haste's AST

of *forever* as JavaScript permits loops to be broken out of at will. The AST, however, does not permit this. The only way to leave a $forever$ statement is to return from the enclosing function.

The ret_f and ret_t both correspond to the JavaScript `return` statement, the distinction only made for the benefit of the optimizer being able to distinguish between the two at need. The *stop* statement fills a similar niche, representing a no-op code flow termination, its JavaScript counterpart simply a blank line. The $tailcall f(...)$ statement morally corresponds to the JavaScript statement `return f(...)`, but has one crucial difference: it uses the trampolining machinery described in section 2.2 to avoid growing the call stack.

The non-terminating statements have a rather more direct correspondence to proper JavaScript. The *switch* statement corresponds directly to JavaScript's `switch`, and variable assignment is the direct equivalent of its JavaScript counterpart as well. Note that there is a slight distinction between the assignment forms $var\ v\ =\ exp\ ;\ ...$ and $exp_0\ =\ exp\ ;\ ...$ in that the former always creates a fresh variable in the current scope, whereas the latter mutates an already existing l-value.

Expressions If the statement language of the AST has some idiosyncrasies when compared to JavaScript, the expression language is quite a bit less restricted. Expressions representing the usual JavaScript constructs – literals, variables, binary operators, array creation and indexing, and function expressions – all correspond directly to their JavaScript namesakes.

Function calls are implemented according to the scheme laid out in section 2.3, but do *not* handle calls to the trampolined functions described in section 2.2. Instead, this is the domain of the f_{tr} trampolined function call, which may be seen as functionally superseding the regular function call in that it is able to deal with calls to any function. It is, however, more expensive than the normal function call, and so the AST provides the faster operation to be used when possible. The creation and evaluation of thunks is implemented as described in 2.1. The expression $thunk(f, upd)$, where f is a nullary function, creates a thunk object with f as its body and its updatable flag set to the value of *upd*. Its counterpart $eval(t)$ evaluates the thunk t as previously described and returns the resulting value.

With the relation between the AST and proper JavaScript established, from this point we will forego any further discussion of this mapping and instead focus on Haste's translation from STG to AST.

4.3 Translating STG into AST

The translation from STG into AST is for the most part relatively straightforward. STG expressions are compiled into their corresponding AST expressions – almost a 1:1 correspondence – with the exception of *let*-bindings and *case* expressions. The general idea of the compilation scheme is to translate an STG expression e into an AST expression $E(e)$, and a *supporting statement continuation* $\lambda cont.S(e, cont)$, which consists of any *switch* statements, variable assignments, and other statements depended on by $E(e)$. When the end of a code path is reached, the continuation is plugged by an appropriate terminating statement: *stop* for the branches of a *switch* statement and top level bindiings, ret_t for thunks, and ret_f for functions. Post-generation, an optimization pass is then run over each function which eliminates unnecessary variable assignments and attempts to reduce the performance impact of the trampolining machinery. The non-trivial optimizations performed by Haste are described further in section 5.

The definitions of E and S are given in tables 2.7 and 2.8 respectively. Due to the comparatively complex nature of *case* expressions, their translation into supporting statement continuations are given separately in table 2.9. This translation in turn makes use of the following definitions:

- $L(l)$ gives the translation of an STG machine literal l into a corresponding AST literal according to the data representations defined in section 3.
- $V(n)$ gives the translation of an STG identifier n into a corresponding AST identifier.
- $upd(t)$ gives the update flag for a thunk as described in section 2.1.
- $prim(op)$ gives the AST implementation of a primitive operation op. Primitive operations are further discussed in section 4.1.
- $fresh(v)$ gives a fresh, unique identifier based on v. This construct is used to introduce new, predictable identifiers for storing the result of a *case* expression.

A Haskell module, represented in STG as a list of bindings, is translated into AST by applying S to each binding:

```
compileModule :: [StgBinding] → [AST]
compileModule = map (λbnd → S(bnd, Stop))
```

This results in a list of AST bindings, which are then optimized and stored as an AST module consisting of identifier–binding pairs. The set of available such modules and a root symbol, corresponding to the `main`

$$
\begin{aligned}
E(l) &= L(l) \\
E(v) &= V(v) \\
E(f\ a_{1..n}) &= \begin{cases} E(f)_{tr}(A(a_1),\ ...) & \text{if } n > 0 \\ eval(f) & \text{otherwise} \end{cases} \\
E(conApp(c,\ a_{1..n})) &= \{id_{tag} :\ tag(c),\ id_1 :\ A(a_1),\ ...\} \\
E(opApp(op,\ a_{1..n})) &= prim(op)(A(a_1),\ ...) \\
E(l@(\lambda().\ e)) &= thunk(fun()\{S(e,\ ret_t\ E(e))\}, upd(l)) \\
E(\lambda v_{1..n}.\ e) &= fun(V(v_1),\ ...)\{S(e,\ ret_f\ E(e)\} \\
E(case\ v@e\ of\ a_{1..n}) &= fresh(v) \\
E(let\ v = e_0\ in\ e_1) &= E(e_1) \\
E(let\_no\_escape\ v = e_0\ in\ e_1) &= E(e_1)
\end{aligned}
$$

$$
A(a) = \begin{cases} L(a) & \text{if literal } a \\ V(a) & \text{otherwise} \end{cases}
$$

Table 2.7: Expression translation

$$
\begin{aligned}
S(case\ v@e\ of\ a_{1..n}, s) &= C(v,\ e,\ a_{1..n},\ s) \\
S(let\ v = e_0\ in\ e_1, s) &= var\ V(v) := E(e_0)\ ;\ S(e_1,\ s) \\
S(let\_no\_escape\ v = e_0\ in\ e_1, s) &= var\ V(v) := E(e_0)\ ;\ S(e_1,\ s) \\
S(\_,\ s) &= s
\end{aligned}
$$

Table 2.8: Statement translation

function of a Haskell program, constitutes the input of the linking process described in section 4.4.

The observant reader will note that the given translation of STG into AST does not make use of all of the ASTs syntactic forms. Some are only introduced by post-generation optimization passes, and some are used extensively by Haste's implementation of various primitive operations but scarcely otherwise.

The treatment – or rather, non-treatment – of the *let_no_escape* also deserves some explanation. Initially, Haste's code generator treats let-no-

$$
\begin{array}{rl}
C(v_0, e_0, a_{1..n}, s) = & S(e_0, \\
& \quad \mathit{var}\ V(v_0) := E(e_0)\ ; \\
& \quad \mathit{switch}(\mathit{tag}(v_0))\ \{ \\
& \quad\quad \mathit{altTag}(a_1) : \mathit{alt}(a_1, v_0), \\
& \quad\quad ... \\
& \quad\quad \mathit{altTag}(a_n) : \mathit{alt}(a_n, v_0) \\
& \quad \}\ ;\ s) \\
\\
\mathit{alt}((\_, v_{1..n}, e), v_0) = & \mathit{var}\ V(v_1) := E(v_0).\mathit{id}_1\ ; \\
& ... \\
& \mathit{var}\ V(v_n) := E(v_0).\mathit{id}_n\ ; \\
& S(e, \mathit{var}\ \mathit{fresh}(v_0) := E(e)\ ;\ \mathit{stop}) \\
\\
\mathit{altTag}((c, \_, \_)) = & \begin{cases} t & \text{if } \mathit{conApp}(t, ...) = c \\ L(c) & \text{otherwise} \end{cases} \\
\\
\mathit{tag}(v) = & \begin{cases} V(v).\mathit{id}_{\mathit{tag}} & \text{if algebraic } v \\ V(v) & \text{otherwise} \end{cases}
\end{array}
$$

Table 2.9: *case* expression translation

escape bindings no different from the standard let bindings. Let-no-escape bindings may be mutually recursive, and so would be slightly tricky to implement efficiently during initial code generation in the absence of an unstructured jump construct. Instead, a post-generation optimization pass attempts to inline local single call functions, including those generated by entering let-no-escape bindings, avoiding unnecessary function call overhead. Those let-no-escape bindings that are not recursive are instead handled by the tail call machinery described in sections 2.2 and 5, as any other bindings. While a more efficient treatment of these bindings is indeed possible, the additional complexity of implementation has so far outweighed the performance benefits.

4.4 Symbol level linking

After code generation is complete, we end up with a program consisting of a list of bindings, one of which is the program's entry point if we are compiling an executable as opposed to a library. However, this program will in all likelihood not be complete: except for the very lowest level building blocks of the standard library, any program will always depend on external code, and thus need to be linked together with its dependencies.

To avoid unnecessarily inflating the size of its final JavaScript output, Haste performs linking on the symbol level, assembling a list of all symbols and their definitions needed to execute the program being linked. Starting from the entry point of the program being linked, the linker looks for references to non-local symbols. Whenever one is found, the symbol is looked up in Haste's library environment and its definition prepended to the list of definitions necessary for program execution. This process is then repeated recursively until no more previously unencountered external symbols are encountered. The list of definitions is then turned into a complete AST program, consisting of the assignments of external definitions to their symbols followed by a call to the program's entry point function.

After the linking stage, a whole program optimization pass is optionally run over the resulting program, to perform the same inlining and other optimizations as were performed on a per function basis, but this time over the whole program.

5 Optimizing JavaScript

Optimization is one area where Haste's guiding philosophy – it's someone else's problem – shines the brightest. As explained in section 1.1, using GHC's intermediate STG representation as our source language gives us a full suite of state of the art optimizations for our Haskell programs, as well their various intermediate representations, essentially for free. We also get the quite powerful framework of *rewrite rules* – the ability to specify equivalences between a source expression and a (hopefully) more efficient but possibly more complex target expression to guide the optimizer – at our disposal. These optimizations apply to Haskell and GHC's intermediate formats in general; there is ample space for further optimization of the generated code, tuning it specifically for execution in the web browser. This space also benefits from Haste's outsourcing principle.

JavaScript is a major application language for a platform where code size is an important factor, owing to the frequent on-demand redownloading of JavaScript programs. Execution speed, while not quite as important due to the current state of processor time being cheap whereas bandwidth is

quite expensive, is also a concern due to the meteoric rise of handheld devices with limited computing power. With this in mind, it comes as no surprise that there exist a wealth of JavaScript-to-JavaScript optimizers, often called *minifiers* due to their focus on reducing code size and thus bandwidth requirements. Haste integrates directly with Google's *Closure compiler*, one advanced such optimizer, adding another full suite of state of the art optimizations to its stable, this time for the generated JavaScript code.

There exists, however, yet another optimization niche which is filled by neither of these approaches: optimizing the resulting JavaScript code using knowledge and assumptions about our source language, runtime system and source program. GHC cannot do it because it does not know it is optimizing for JavaScript code generation and Closure cannot do it because it knows nothing about Haskell, STG, laziness or the controlled appearance of effects; neither knows anything about Haste's runtime system. These optimizations consist mostly of relatively uninteresting cleanup: removing unnecessary assignments, shrinking expressions to smaller or more efficient equivalents, and so on. Some optimizations performed are larger in scope, however, and deserve a more in-depth treatment.

5.1 Tail call elimination

As explained in section 2.2, JavaScript does not support proper tail calls, and Haste thus needs to make use of trampolining to support general tail calls. The procedure to turn a "normal" function call in tail position into a proper tail call is trivial. We first recursively turn all *intermediate assignments* – variable assignments of the form *var* $x = e$; *return* x – into substitutions, replacing any such assignments by *return* e. Then, all occurrences of *return* $f(...)$ are converted into a special syntactic form *tailcall* $f(...)$, which returns a continuation object performing the call to f for evaluation by the trampolining instrumentation as described in section 2.2.

Loop transformation However, in the more specific – and quite common – case of simply tail recursive functions, we can make the tail recursion quite a bit faster by employing JavaScript's looping constructs. If the body b of a function f contains at least one occurrence of $tailcall f(x_0, ..., x_n)$, it is simply tail recursive and we can replace the body of f with a loop. This loop executes b but replaces the tail call to $f(a_0, ..., a_n)$ with a series of assignments *var* $x_0 = a_0$, ..., $x_n = a_n$ followed by a *continue* statement. Figures 2 and 3 give an example of a tail recursive multiplication function before and after this loop transformation respectively.

```
function mul(x, y, accumulator) {
  if(y === 0) {
    return accumulator;
  } else {
    tailcall(mul(x, y-1, accumulator+x));
  }
}
```

Figure 2: Tail recursive multiplication function

```
function mul(x, y, accumulator) {
  while(true) {
    if(y === 0) {
      return accumulator;
    } else {
      y = y - 1;
      accumulator = accumulator + x;
      continue;
    }
  }
}
```

Figure 3: Tail loop-transformed multiplication function

Loop transformation in the presence of closures However, the loop transformation optimization fails subtly in the case where the function body contains function closures. In JavaScript, closures capture variables by reference. As we recall, function arguments may be mutated between each "invocation" of a tail loop transformed function. If a closure is created in the loop body *b* which captures one of these mutating variables, those variables will most likely have mutated between the time the closure is created and the time it is entered!

Haste makes use of the fact that JavaScript function arguments are always passed by value to solve this problem, wrapping each iteration of the loop in an anonymous function taking all of the mutating variables as arguments. The result is that each mutating variable is *explicitly copied* for each iteration of the loop. Each created closure thus captures its own copies of the mutating variables, instead of capturing the mutable references it would have had without this explicit copying.

This essentially creates a local, specialized trampoline for each affected function. While this is more expensive than the pure loop optimization, the

```
function mul(x0, y0, accumulator0) {
  while(true) {
    var result = (function(x, y, accumulator) {
      if(y === 0) {
        return accumulator;
      } else {
        y0 = y - 1;
        accumulator0 = thunk(function(){return eval(accumulator) + x}, true);
        return __continue;
      }
    })(x0, y0, accumulator0);
    if(result !== __continue) {
      return result;
    }
  }
}
```

Figure 4: Closure-correct loop transformation

specialization and comparatively fewer levels of indirection still makes it a cheaper construct than the general trampoline. Figure 4 gives an example of this transformation for a version of the `mul` function from figure 2 which is lazy in its accumulator, necessitating explicit argument copying.

5.2 Mitigating trampolining overheads

While loop transformation of simple recursive functions is all well and good, the general problem still remains: trampolining is slow. To make matters worse, we invoke our trampoline for *all* function calls, even though the vast majority may not even need it! Fortunately, it is possible to eliminate the trampolining instrumentation from many call sites where it is unnecessary, as well as convert some slow tail calls into fast "normal" calls without using any additional stack frames.

Eliminating acyclic tail calls The point of tail call elimination is to allow a chain of tail calls of arbitrary length to execute in constant space. By finding finite chains of trampolined tail calls up to a certain length and converting them into normal function calls we can reduce the trampoline instrumentation overhead while maintaining a constant upper bound on the amount of stack space consumed by the tail call. This optimization, proposed by Loitsch and Serrano [30], is implemented in Haste as follows.

We begin by finding all functions which are guaranteed to never perform a tail call and convert all tail calls to those functions into normal, fast,

function calls. In this way we eliminate the overhead of creating and calling a trampoline object for those functions while guaranteeing that no chain of tail calls will grow the call stack by more than at most one frame.

Repeating this procedure n times, we can eliminate all tail call chains of length n or less while bounding call stack growth to n frames. After we've finished converting tail calls into normal calls, we can remove the trampolining instrumentation from all call sites where the callee is guaranteed to never make a tail call, completely removing the trampolining overhead from a significant number of call sites.

In practice, while many statically known function calls can be detrampolined, the amount of actual tail calls eliminated by this optimization drops off sharply as n increases. With the aggressive inlining performed by GHC, practical programs usually end up with few acyclic tail call chains, and few of those are longer than one or two calls. Haste uses a value of three for n, which bounds call stack growth to three stack frames. This eliminates virtually all finite tail call chains, and carries an extra overhead of only a single stack frame, compared to the tail call instrumentation which itself has an overhead of two additional stack frames per call chain.

Trading space for speed Even after eliminating tail calls statically determined to be unnecessary, there is still room for improvement. Not only are actual cyclic call chains unaffected by the above optimizations, but so are functions whose identity cannot be statically determined as well – a common occurrence in a higher order language. To improve the performance of the general tail call machinery, we employ an optimization described by Schinz and Odersky to trade stack space for speed [50].

The runtime system keeps track of a *chain counter*, which records the length of the current chain of tail calls. While this counter is below a certain threshold, *no tail calls are made at all*. Any tail call made below this threshold will cause the chain counter to be incremented by one and a normal, fast, stack-growing function call to be made. When a tail call is made after this threshold is reached, however, the normal tail call procedure comes into play and a trampoline object is returned to the caller, which returns it back to its own caller and so on, until the object reaches the trampolining instrumentation at the bottom of the call chain. Here, the chain counter is reset to 0, and the trampoline object invoked.

This way, we can use fast function calls for n tail calls in a row, only resorting to the slow trampolining machinery once every n tail calls. This allows each chain of consecutive tail calls to grow the call stack by at most n additional stack frames, amortizing the cost of the trampolining instrumentation over the calls. Regular trampolining, where the call stack is

not allowed to grow at all, can be seen as a special case of this optimization, where n is equal to 1.

5.3 Reducing indirections

In vanilla GHC, numeric types are implemented using one level of indirection. A value of type `Int` will thus be stored on the heap as a pointer to a thunk object, which in turn will have a reference to either a computation which produced a value of type `Int`, or to an actual machine level integer. This is necessitated by the implementation strategy described by Marlow and Peyton Jontes [36]. However, as discussed in section 2.1, As Haste uses a slightly different approach, inspecting thunks from the outside rather than unconditionally entering them, we are able to represent values of some types using one less level of indirection than vanilla GHC.

Haste has the concept of *newtype-like* types, conservatively defined to consist of all types with a single data constructor, that in turn has only a single, unlifted, argument. All such types are represented in Haste as though they were **`newtype`**s: the representation of such a type constructor application `C x` is the representation of its argument `x`. Crucially, this definition includes all the basic numeric types. This allows us to get rid of a significant amount of boxing and unboxing operations, as boxed and unboxed numeric types now share the same representation. Furthermore – and more importantly – this also reduces pressure on the garbage collector. While boxed numeric types may have relatively little overhead in vanilla GHC, this overhead is quite significant for Haste, which has to resort to a JavaScript representation of algebraic types. While most computationally heavy Haskell code usually ends up unboxed by GHC's optimizer, removing a large chunk of the overhead caused by boxing and unboxing still produces code with smaller footprints for both code size and memory consumption.

6 Performance evaluation and discussion

As indicated at the beginning of this chapter, reducing code size is one of the primary motivations for the Haste compiler. However, program execution time is also an important factor. This section presents a series of benchmarks, taken from the *nofib* [40] benchmark suite, to measure Haste's performance compared to the GHCJS compiler as well as the performance impact made by the optimizations described in section 5. The programs were selected from the compatible ones in the suite – several benchmarks cannot be completed by Haste or GHCJS due to reliance on missing native libraries – to give a balanced view of the code size, raw computation performance, and performance under GC pressure.

The Haste programs were compiled using Haste version 0.5.3 with the `--opt-all` flag and, for the minified versions, the `--opt-minify` flag as well which calls the Closure compiler with the *ADVANCED_OPTIMIZATIONS* compilation flag on the generated output. The GHCJS programs were compiled with the latest development version of GHCJS as of October 2015, using the `-02` flag for optimizations. The resulting JavaScript programs were executed using version 4.1.1 of the Node.js interpreter.

6.1 Relative performance of the Haste compiler

We measure the performance of the latest version of Haste as compared to the latest development version of the GHCJS compiler on two counts: code size, and execution speed. As GHCJS is considered by many to be the de facto standard web-targeting Haskell compiler and the state of the art in Haskell to JavaScript compilation, it is the natural target of performance comparisons. GHCJS also uses STG code produced by GHC as its input format, but uses a completely different compilation scheme, compiling programs into continuation-passing style, and a more involved runtime system [39]. Comparing against GHCJS thus gives an opportunity to evaluate the relative performance of the two approaches to JavaScript compilation without interference from, for example, compiler frontends of differing quality.

Execution time The results of the speed benchmarks are given in table 2.10. All run times are given in seconds. The *Haste-min* columns gives the execution time of the relevant program compiled with Haste and minified using the Closure compiler, with the *ADVANCED_OPTIMIZATIONS* compilation flag. The corresponding execution times are not given for GHCJS entries, as the GHCJS programs give incorrect results when minified.

For this set of benchmarks, Haste holds a significant advantage in execution speed across the board, with the *binary-trees* and *queens* benchmarks being more than twice as fast when compiled with Haste than with GHCJS. This may be attributed to Haste having a relatively straightforward and idiomatic implementation of function calls, whereas GHCJS CPS-transformed code is quite heavily trampolined. The advantage is less dramatic, but still significant, for the rest of the programs. The smallest difference in execution speed can be seen in the *power* benchmark, which is dominated by time spent computing over integers of arbitrary size. As Haste and GHCJS outsource this particular functionality to the same JavaScript library, the relatively small difference in execution speed comes as no surprise.

For this set of benchmarks, minification seems to have a relatively neglibible impact on execution times, with only the *circsim* benchmark

Program	Haste	Haste (min.)	GHCJS	Diff.
binary-trees (n=16)	5.1 s	5.5 s	12.3 s	2.4
queens (n=12)	2.7 s	2.6 s	6.5 s	2.4
integrate (n=100k)	1.6 s	1.5 s	3.2 s	2.0
power (n=25)	0.5 s	0.5 s	0.8 s	1.6
circsim (8 bits, 100 cycles)	1.1 s	1.0 s	1.6 s	1.5

Table 2.10: Code execution speed

Program	Haste	Haste (min.)	GHCJS	GHCJS (min.)
binary-trees	157 KiB	86 KiB	1336 KiB	349 KiB
queens	82 KiB	23 KiB	1003 KiB	226 KiB
power	100 KiB	37 KiB	1200 KiB	295 KiB
anna	592 KiB	375 KiB	3427 KiB	1166 KiB

Table 2.11: Emitted code size

standing out with its 10 % shorter execution time compared unminified counterpart. Interestingly, the *binary-trees* program actually runs about 10 % *slower* when minified, showing that minification is not always beneficial to execution speed.

Code size The size of its generated code becomes particularly interesting as reduced code size is a main motivator for Haste. Several of the programs from the *nofib* benchmark suite were compiled with Haste as well as with GHCJS, and the size of their respective outputs were inspected. The results are given in table 2.11.

Haste emerges as the clear winner of the code size benchmarks, with a larger margin than for the speed benchmarks. Depending on the program, the code generated by GHCJS is larger by a factor of 6 to 10, although the difference shrinks with increased program size. Both Haste and GHCJS programs seem to respond very well to minification. Although minification presently breaks the GHCJS programs, either changing their semantics or causing them to abort with an error message, its effect on the size of the

generated code is significant, and is likely to remain so were the problems resulting in broken code to be fixed.

6.2 Performance of tail call optimizations

The trampolining optimizations described in section 5 – loop transformation, acyclic tail call elimination and tail chain counting – all in all have a significant impact on performance, albeit in different circumstances. The most generally applicable optimization is the acyclic tail call elimination, which removes unnecessary tail calls and trampolining during a whole program optimization pass. Turning this optimization off results in a 10 % slowdown across the entire set of benchmarks.

The utility of the loop transformation optimization is less universal, but its effect on execution speed can be more pronounced where the optimization applies. For the *queens* benchmark, disabling the loop transformation optimization results in a 25 % slowdown. For the *binary-trees* benchmark, disabling this optimization led to a slowdown of more than 10 %. The other benchmarks saw no significant performance improvements – or penalties – as they are relatively light on the tail recursion.

Together with the loop transformation optimization, the tail chain counter optimization becomes highly situational: quite many tail calling functions compile into simple loops, which are covered by the loop transformation. Consequently, the only benchmarks in which this optimization made a difference one way or the other – or even appeared in the generated source – were the *binary-trees* and *queens* benchmarks. However, once this optimization kicks in it is highly effective: for *binary-trees* disabling the tail chain counter resulted in a 30 % slowdown. For *queens*, disabling the optimization resulted in a 30 % slowdown as well, but only if the tail loop transformation was also disabled. Taken together, this indicates that while there is quite some overlap between the loop transformation and the tail chain counter optimizations, the tail chain counter provides a useful mitigation for the tail recursive cases which are not covered by the simple loop.

The lower applicability of these two optimizations is not surprising: the acyclic tail call elimination reduces the general overhead of trampolining on *non*-tail recursive functions, which appear in generous quantities in virtually any program. Tail recursion, while quite common in functional programs, is not nearly as ubiquitous.

Program	Classy	Anonymous	Arrays
binary-trees (n=16)	5.1 s	4.9 s	5.7 s
queens (n=12)	2.7 s	5.9 s	8.0 s
integrate (n=100k)	1.6 s	2.0 s	2.6 s
power (n=25)	0.5 s	0.5 s	0.6 s
circsim (bits=8, cycles=100)	1.1 s	2.0 s	3.2 s

Table 2.12: Performance comparison of ADT representation candidates

6.3 ADT representation: objects versus arrays

Haste's abstract syntax makes it relatively straightforward to experiment with different data representations for algebraic data types by swapping out the JavaScript serialization of the data constructor and accessor primitives. In order to evaluate the performance of the three different ADT representations discussed in section 3 – classy objects, anonymous objects and arrays – the benchmarks used throughout this section were compiled and run with all three different representations. The results are listed in table 2.12.

Judging by these benchmarks, the classy objects approach is significantly faster than the other two representations, at least on the V8 virtual machine used by Node.js. While the *binary-trees* benchmark is about 4 % slower, the substantial difference in execution speed for the *queens*, *integrate* and *circsim* programs more than makes up for this slight deficiency. Comparing with the GHCJS tests from the previous section, Haste does quite well on the *binary-trees* benchmark with either representation. Similarly, the *power* benchmark is dominated by time spent in external libraries, which may explain the relative lack of difference in performance for this benchmark.

Although not presented in the table, the classy approach also proved more amenable to minification than the other approaches, seeing the relatively encouraging performance improvements described in the previous section. Meanwhile, minification impact on execution speed was relatively hit and miss for anonymous objects and arrays, having a negative impact on execution speed as often as a positive one.

6.4 Summary

In this chapter we have presented the design and implementation of a web-targeting Haskell compiler. We have discussed the design choices

made in the design of the compilation scheme as well as the runtime system and chose data representations, and contrasted them with the existing state of the art for performance as well as code footprint. While none of the techniques used in the Haste compiler are entirely novel in and of themselves, the Haste compiler is to our knowledge the first combined application of these techniques – the high level translation scheme and data representation, the extensive trampolining optimizations, and the fine grained linking – in a JavaScript-targeting compiler for a lazy functional language.

As shown in our performance evaluation, the application of these techniques results in a compiler which produces code which is both faster and smaller than the current state of the art, by factors of two and six respectively. To our knowledge, ours is the first evaluation of implementation techniques for a web-targeting compiler for a lazy functional language. Thanks to the shallow runtime system and high level translation scheme our approach combines high performance with simple interoperability, the applications of which are explored in chapter 3.

Interoperating with JavaScript

This chapter is based on the paper *Foreign Exchange at Low, Low Rates* [20], which is under consideration for publication at IFL '15, and describes the Haste compiler's foreign function interface.

In section 1 we describe the challenges of interfacing with high level foreign code using the vanilla foreign function interface of GHC in a web environment, and propose a list of qualities we would like to see in a foreign function interface for a web environment. In section 2 we present the *Haste.Foreign* foreign function interface which has all the properties set forth in section 1, and give its implementation for the Haste compiler. In sections 3 and 4 we propose several extensions to the basic interface which extends its scope and improves on its performance, and in section 5 we evaluate the performance of Haste.Foreign compared to the vanilla foreign function interface of GHC. In section 6 we discuss the limitations of Haste.Foreign, as well as possible means of lifting or working around said limitations.

1 Background

Interfacing with other languages is one of the more painful aspects of modern day Haskell development. Consider figure 5, taken from the standard libraries of GHC; a piece of code to retrieve the current time [57]. A relatively simple task, yet its implementation is surprisingly complex.

This code snippet is more akin to thinly veiled C code than idiomatic, readable Haskell; an unfortunate reality of working with the standard foreign function interface.

While Haste initially made use of the conventional Foreign Function Interface extension [8] to interface with its browser target environment, this presented certain difficulties. The modern web browser environment is highly reliant on callback functions and complex data types, none of which are trivial to pass through the FFI, making browser-interfacing Haste code relatively clunky and byzantine.

```
data CTimeval = MkCTimeval CLong CLong

instance Storable CTimeval where
        sizeOf _ = (sizeOf (undefined :: CLong)) * 2
        alignment _ = alignment (undefined :: CLong)
        peek p = do
                s   ← peekElemOff (castPtr p) 0
                mus ← peekElemOff (castPtr p) 1
                return (MkCTimeval s mus)
        poke p (MkCTimeval s mus) = do
                pokeElemOff (castPtr p) 0 s
                pokeElemOff (castPtr p) 1 mus

foreign import ccall unsafe "time.h gettimeofday"
   gettimeofday :: Ptr CTimeval → Ptr () → IO CInt

getCTimeval :: IO CTimeval
getCTimeval = with (MkCTimeval 0 0) $ λptval → do
  throwErrnoIfMinus1_ "gettimeofday" $ do
    gettimeofday ptval nullPtr
  peek ptval
```

Figure 5: Foreign imports using the vanilla Foreign Function Interface

To rectify this situation, we construct a more expressive FFI on top of the old one. We decompose interactions with the host environment into its constituent parts: marshalling arguments into the target language, performing the actual foreign call, and finally marshalling the results back into Haskell. We implement these parts in Haskell itself to the extent possible, only reaching out to the host environment through the FFI for our lowest level building blocks. The result is a foreign function interface which to a high degree automates the tedium involved in communicating with a foreign environment.

Traditionally, Haskell programs have used the Foreign Function Interface extension to communicate with other languages. This works passably well in the world of native binary programs running on bare metal, where C calling conventions have become the de facto standard of foreign data interchange. The C language has no notion of higher level data structures or fancy data representation, making it the perfect lowest common denominator interlingua for language to language communication: there is no ambiguity or clash between different languages' built-in representation of various higher level data structures, as there simply *are* no higher level data structures.

The same properties that make Haskell's traditional foreign function

interface a good fit for language interoperability make it undesirable as a vehicle for interfacing with the web-targeting code: the guest language commonly relies on the browser environment for a large part of its runtime, and internally uses many of its native high level data structures and representations, making the forced low level representations of the vanilla foreign function interface an unnecessary obstacle rather than a welcome common ground for data interchange.

With this background, we believe that low level interfaces such as the vanilla FFI are not ideally suited to the domain of functional languages targeting the web browser and other high level environments. More specifically, we would like a foreign function interface for this domain to have the following properties:

- The FFI should automatically take care of marshalling for any types where marshalling is defined, without extra manual conversions or other boilerplate code.
- Users should be able to easily define their own marshalling schemes for arbitrary types.
- The FFI should allow importing arbitrary host language code, not just named, statically known functions.
- Finally, the FFI should be easy to implement and understand, ideally being implementable without compiler modifications, portable across guest language dialects and host environments.

Making this list a bit more concrete in the form of an example, we would like to write high level code like that in figure 6, without having to make intrusive changes to our Haskell compiler.

Contrasting this with the standard FFI code from figure 5:

- The low level C types are gone, replaced by a more descriptive record type, and so is the `peek`ing and `poke`ing of pointers.
- The imported function arrives "batteries included", on equal footing with every other function in our program. No extra scaffolding or boilerplate code is necessary.
- Whereas the code in figure 5 had to import the `gettimeofday` system call by name, its actual implementation given elsewhere, we have actually *implemented* its JavaScript counterpart at the location of its import, without having to resort to external stubs.

```
data UTCTime = UTCTime {
    secs  :: Word,
    usecs :: Word
  } deriving Generic

instance FromAny UTCTime

getCurrentTime :: IO UTCTime
getCurrentTime =
  host "function() {\\
          \\var ms = new Date().getTime();\\
          \\return {secs:  ms/1000,\\
          \\        usecs: (ms % 1000)*1000};}"
```

Figure 6: Foreign imports using our FFI

In section 2, we present the design and implementation of an interface fulfilling the above criteria. The basic interface is implementable using plain Haskell '98 with only the Foreign Function Interface extension, and is extensible by the user in the types of data which can be marshalled between Haskell and host language, as well as in how those types are marshalled. It allows for context dependent sanity checking of incoming data from the host language, improving the safety of foreign functions.

While designed and described for web-targeting Haskell dialects in general and the Haste compiler in particular, the interface is applicable outside the web domain as well, and the implementation we give is valid for dynamically typed host languages that support garbage collection, first class functions, and a construct for dynamic code evaluation at runtime such as the `eval` function of JavaScript, Python, PHP, and others.

To be clear, the idea of a higher level foreign function interface is by no means novel in itself; there already exists a large body of work in this problem domain, solving several of the problems of figure 5, which is used here as an example mainly to establish the baseline for foreign function interfaces.

We discuss these related approaches in section 2 of chapter 1, contrasting them with our approach. To our knowledge, our solution is the first to address all of the aforementioned criteria however. In particular, we are not aware of any other FFI framework that can be implemented entirely without compiler modifications.

2 An FFI for the modern web

2.1 The interface

This section describes the programmer's view of our interface and gives examples of its usage. The Haskell formulation of the interface is given in figure 7.

As the main purpose of a foreign interface is to shovel data back and forth through a rift spanning two separate programming worlds, it makes sense to begin the description of any such interface with one central question: what data can pass through the rift and come out on the other side still making sense?

The class of data fulfilling this criterion is embodied in an abstract `HostAny` data type, inhabited by host-native representations of arbitrary Haskell values. A data type is then considered to be marshallable if and only if it can be converted to `HostAny` and back again.

Having established the class of types that can be marshalled, we can now give a meaningful definition of *importable* functions: a function can be imported from the host language into our Haskell program if and only if:

- all of its argument types are convertible into `HostAny`;
- its return type is convertible *from* the host-native `HostAny`; and
- its return type resides in the `IO` monad, accounting for the possibility of side effects in host language functions.

These definitions give rise to a workflow for interacting with host language code:

- define the appropriate `ToAny` and `FromAny` instances for any custom types, either automatically using the generic default instances as showcased by our motivating example in figure 6, or by defining them manually if a particular host language representation is desired; then
- import arbitrary host language symbols or expressions over any set of types instantiating `ToAny` or `FromAny`, using the `host` function.

We let the classic "hello, world" example illustrate the import of simple host language functions using the interface described in figure 7:

```
hello :: String → IO ()
hello = host "name ⇒ alert('Hello, ' + name);"
```

To further illustrate how this interface can be used to effortlessly import even higher order foreign functions, we have used our library to implement

```
type HostAny

class ToAny a where
  toAny :: a → HostAny
  default toAny :: (GToAny (Rep a), Generic a)
                ⇒ a → HostAny

class FromAny a where
  fromAny :: HostAny → IO a
  default fromAny :: (GFromAny (Rep a), Generic a)
                  ⇒ HostAny → IO a

class Import f
instance (ToAny a, Import b) ⇒ Import (a → b)
instance FromAny a           ⇒ Import (IO a)

-- Instances for functions and basic types
instance FromAny Int
instance ToAny Int
...
instance Import f ⇒ FromAny f
instance (FromAny a, Exportable b) ⇒ ToAny (a → b)
instance ToAny a ⇒ ToAny (IO a)

host :: Import f ⇒ String → f
```

Figure 7: The programmer's view of our interface

bindings to JavaScript *animation frames* for the Haste compiler, a mechanism whereby a user program may request the browser to call a certain function before the next repaint of the screen occurs:

```
type Time = Double
newtype FrameHandle = FrameHandle HostAny
  deriving (ToAny, FromAny)

requestFrame :: (Time → IO ()) → IO FrameHandle
requestFrame = host "window.requestAnimationFrame"

cancelFrame :: FrameHandle → IO ()
cancelFrame = host "window.cancelAnimationFrame"
```

The resulting code is straightforward and simple, even though it performs the rather non-trivial task of importing a foreign higher order function, automatically converting user-provided Haskell callbacks to their JavaScript equivalents.

In the rest of section 2, we give an implementation of the basic Haskell '98 interface for the Haste compiler. We then extend it with features requir-

ing some extensions to Haskell '98 – most notably generics and default signatures – in section 4, to arrive at the complete interface presented here.

2.2 Implementing marshalling

As usual in the functional world, we ought to start with the *base case*: implementing marshalling for the basic primitive types that lie at the bottom of every data structure.

This is a simple proposition, as this is the forte of the vanilla foreign function interface.

```
foreign import ccall intToAny :: Int → HostAny
foreign import ccall anyToInt :: HostAny → IO Int

instance ToAny Int where toAny = intToAny
instance FromAny Int where fromAny = anyToInt
...
```

We might also find a `HostAny` instance for `ToAny` and `FromAny` handy. Of course, `HostAny` already being in its host language representation form, the instances are trivial.

```
instance ToAny HostAny where toAny = id
instance FromAny HostAny where FromAny = return
```

However, if passing simple values was all we wanted to do, then there would be no need to look any further than the vanilla foreign function interface. We must also provide some way of combining values into more complex values, to be able to represent lists, record types and other conveniences we take for granted in our day to day development work. But how should these values be combined? Depending on our host language, we may have different primitive data structures at our disposal.

Fortunately, JavaScript, as well as virtually any other language for which our interface is implementable as described in section 1, support two basic aggregate types, which are sufficient to represent values of any type: arrays and dictionaries.

For the sake of brevity, we assume that we have access to two functions `arrToList :: FromAny a ⇒ HostAny → [a]` and `listToArr :: ToAny a ⇒ [a] → HostAny` which are used to implement the `FromAny` and `ToAny` instances respectively for lists; they are trivial to implement either in Haskell using the vanilla foreign function interface to gradually build a list of `HostAny` values, or on the host language side exploiting knowledge of the compiler's data representation.

For dictionaries, the conversion is not as clear-cut. Depending on the data we want to convert, the structure of our desired host language representation of two values may well be different even when their client language representations are quite similar, or even identical. Hence, we need to

```
foreign import ccall newDict :: IO HostAny
foreign import ccall set :: HostAny → HostString → HostAny → IO ()
foreign import ccall get :: HostAny → HostString → IO HostAny

mkDict :: [(String, HostAny)] → HostAny
mkDict xs = unsafePerformIO $ do
  d ← newDict
  mapM_ (λk v → set d (toHostString k) v) xs
  return d

getMember :: FromAny a ⇒ HostAny → String → IO a
getMember dict key =
  get dict (toHostString key) >>= fromAny
```

Figure 8: Dictionary manipulation

put the power over this decision into the hands of the user, providing functionality to build as well as inspect user-defined dictionaries.

We will need three basic host language operations: creating a new dictionary, associating a dictionary key with a particular value, and looking up values from dictionary keys. From these we construct two functions to marshal compound Haskell values to and from dictionaries: `mkDict` and `getMember`, as shown in figure 8.

This gives us the power to represent any composite or primitive data type with user-defined dictionary keys. Figure 9 shows a possible marshalling for sum and product types using the aforementioned dictionary operations.

It is worth noting that the implementation of `getMember` is the reason for `fromAny` returning a value in the `IO` monad: foreign data structures are rarely, if ever, guaranteed to be immutable and looking up a key in a dictionary is effectively following a reference, so we must perform any such lookups at a well defined point in time, lest we run the risk of the value being changed in between the application of our marshalling function and the evaluation of the resulting thunk.

2.3 Importing functions

Implementing our `host` function turns out to be slightly trickier than marshalling data between environments. The types of our imported functions need to differ depending on the arity of the imported host language code. This necessitates `host` returning some variadic function. Fortunately, there is a well known trick to accomplish this which uses an inductive class instance to successively build up a list of arguments over repeated function

```
instance (ToAny a, ToAny b) ⇒
         ToAny (Either a b) where
  toAny (Left a)  = mkDict [("tag",  toAny "left"),
                            ("data", toAny a)]
  toAny (Right b) = mkDict [("tag",  toAny "right"),
                            ("data", toAny b)]

instance (FromAny a, FromAny b) ⇒
         FromAny (Either a b) where
  fromAny x = do
    tag ← getMember x "tag"
    case tag of
      "left"  → Left  <$> getMember "data"
      "right" → Right <$> getMember "data"

instance (ToAny a, ToAny b) ⇒ ToAny (a, b) where
  toAny (a, b) = toAny [toAny a, toAny b]

instance (FromAny a, FromAny b) ⇒
         FromAny (a, b) where
  fromAny x = do
    [a, b] ← fromAny x
    (,) <$> fromAny a <*> fromAny b
```

Figure 9: Sums and products using lists and dictionaries

applications, and a base case instance to perform some computation over said arguments after the function in question has been fully applied [1]. In the case of the `host` function, that computation would be applying a foreign function to said list of arguments.

This suggests the following class definition.

```
type HostFun = HostAny

class Import f where
  import_ :: HostFun → [HostAny] → f
```

For our purposes, the base case is a nullary computation in the IO monad. The list of arguments is converted from a list to a host language array in order to squeeze it through the vanilla foreign function interface, and the value we get back is marshalled back into a proper Haskell value:

```
foreign import ccall apply :: HostFun → HostAny → IO HostAny

instance FromAny a ⇒ Import (IO a) where
  import_ f args = apply f (toAny args) >>= fromAny
```

Note the use of a foreign import in our base case. As the application of

a foreign function to a foreign list of foreign arguments is clearly, well, a *foreign* matter, we must call out to the host language for this final step. This function is specific to the host language in use. A possible implementation of `apply` for a JavaScript host environment may look as follows:

```
(f, args) ⇒ f.apply(null, args)
```

The inductive case is not much more complex: we only need to marshal a single argument and recurse.

```
instance (ToAny a, Import b) ⇒ Import (a → b) where
  import_ f args = λarg → import_ f (toAny arg : args)
```

With this, we have all the building blocks required to implement the `host` function. With all the hard work already done, the implementation is simple. For the sake of brevity, we assume the existence of a host language specific `HostString` type, which may be passed as an argument over the vanilla foreign function interface, and a function `toHostString :: String → HostString`.

```
foreign import ccall eval :: HostString → HostFun

host :: Import f ⇒ String → f
host s = import_ f []
  where f = eval (toHostString s)
```

The foreign `eval` import brings in the host language's evaluation construct. Recall that one requirement of our method is the existence of such a construct, to convert arbitrary strings of host language code into functions or other objects.

3 Optimizing for safety and performance

While the implementation described up until this point is more or less feature complete, its non-functional properties can be improved quite a bit if we allow ourselves to stray from the tried and true, but slightly conservative, path of pure Haskell '98.

Aside from implementation specific tricks – exploiting knowledge about a particular compiler's data representation to optimize marshalling, or even completely unroll and eliminate some of the basic interface's primitive operations, for instance – there are several general optimizations we can apply to significantly enhance the performance and safety of our interface.

3.1 Eliminating argument passing overheads

The performance-minded reader may notice something troubling about the implementation of `import_`: the construction of an intermediate list of

```
{-# NOINLINE [0] host' #-}
host' :: FromAny a ⇒ HostFun → [HostAny] → IO a
host' f args = apply f (toAny args) >>= fromAny

instance FromAny a ⇒ Import (IO a) where
  host = host'

foreign import ccall apply0 :: HostFun → IO HostAny
foreign import ccall apply1 :: HostFun → HostAny → IO HostAny
foreign import ccall apply2 :: HostFun → HostAny → HostAny → IO HostAny
...

{-# RULES
  "apply0" [1] ∀f. host' f [] =
               apply0 f >>= fromAny
  "apply1" [1] ∀f a. host' f [a] =
               apply1 f a >>= fromAny
  "apply2" [1] ∀f a b. host' f [b,a] =
               apply2 f a b >>= fromAny
  ...
 #- }
```

Figure 10: Specializing the `host` base case

arguments. Constructing this intermediate list only to convert it into a host language suitable representation which is promptly deconstructed as soon as it reaches the imported function takes a lot of work. Even worse, this work does not provide any benefit for the task to be performed: applying a foreign function.

By the power of *rewrite rules* [44], we can eliminate this pointless work in most cases by specializing the `host` function's base case instance for different numbers of arguments. In addition to the general `apply` function we define a series of `apply0`, `apply1`, etc. functions, one for each arity we want to optimize function application for. The actual specialization is then a matter of rewriting `host` calls to use the appropriate application function.

Figure 10 gives a new implementation of the base case of the `Import` class which includes this optimization, replacing the one given in section 2.

3.2 Preventing code injection

Meanwhile, the *safety-conscious* reader may instead be bristling at the thought of executing code contained in something as egregiously untyped and untrustworthy as a common string. Indeed, by allowing the conversion of arbitrary strings into functions, we're setting ourselves up for cross-site

scripting and other similar code injection attacks!

While this is indeed true in theory, in practice, accidentally passing a user-supplied string to the `host` function, which in normal use ought to occur almost exclusively on the top level of a module, is a quite unlikely proposition. Even so, it could be argued that if it is possible to use an interface for evil, its users almost certainly will at some point.

Fortunately, the recent 7.10 release of the GHC compiler gives us the means to eliminate this potential pitfall. The *StaticPointers* extension, its first incarnation described by Epstein et al [23], introduces the `static` keyword, which is used to create values of type `StaticPtr` from closed expressions. Attempting to turn any expression which is not known at compile time into a `StaticPtr` yields a compiler error.

Implementing a `safe_host` function which can not be used to execute user-provided code becomes quite easy using this extension and the basic `host` function described in section 2, at the cost of slightly more inconvenient import syntax:

```
safe_host :: Import f ⇒ StaticPtr String → f
safe_host = host . deRefStaticPtr

safe_hello :: IO ()
safe_hello = safe_host static "() ⇒ alert('Hello, world!')"
```

3.3 Eliminating `eval`

Relying on `eval` to produce our functions allows us to implement our interface in pure Haskell '98 without modifying the Haskell compiler in question, making the interface easy to understand, implement and maintain. However, there are reasons why it may be in the implementor's best interest to forgo a small bit of that simplicity.

The actual call to `eval` does not meaningfully impact performance: it is generally only called once per import, the resulting function object cached thanks to lazy evaluation.[1] However, its dynamic nature *does* carry a significant risk of interfering with the ability of the host language's compiler and runtime to analyse and optimize the resulting code. As discussed in section 5, this effect is very much in evidence when targeting the widely used V8 JavaScript engine.

In the JavaScript community, it is quite common to run programs through a *minifier* – a static optimizer with focus on code size – before deployment. Not only do such optimizers suffer the same analytical difficulties as the language runtime itself from the presence of dynamically

[1] The main reason for `eval` getting called more than once being unwise inlining directives from the user.

evaluated code, but due to the heavy use of renaming often employed by minifiers to reduce code size, special care needs to be taken when writing code that is not visible as such to the minifier: code which is externally imported or, in our case, locked away inside a string for later evaluation.

Noting that virtually every sane use of our interface evaluates a *static* string, a solution presents itself: whenever the `eval` function is applied to a statically known string, instead of generating a function call, the compiler splices the contents of the string verbatim into the output code instead.

This solution has the advantage of eliminating the code analysis obstacle provided by `eval` for the case when our imported code is statically known (which, as we noted before, is a basic sanity property of foreign imports), while preserving our library's simplicity of implementation. However, it also has the *dis*advantage of requiring modifications to the compiler in use, however slight, which increases the interface's overall complexity of implementation.

4 Putting our interface to use

While the interface described in sections 2 and 3 represents a clear raising of the abstraction layer over the vanilla foreign function interface, it is still lacking some desirable high level functionality: marshalling of arbitrary functions and generic data types.

In this section we demonstrate the flexibility of our interface by implementing this functionality on top of it.

4.1 Dynamic function marshalling

Dynamic imports One appealing characteristic of our interface is that it makes the marshalling of functions between Haskell and the host language easy. In the case of passing host functions into Haskell, the `import_` function used to implement `host` has already done the heavy lifting for us. Only adding an appropriate `FromAny` instance remains.

Due to the polymorphic nature of functions, however, we must resort to using some language extensions to get the type checker to accept our instance: overlapping instances, flexible instances, and undecidable instances. Essentially, the loosened restrictions on type class instances allow an `Import` instance to act as a synonym for `FromAny`, allowing host language functions to return functions of any type admissible as an import type by way of the `host` function.

```
instance Import a ⇒ FromAny a where
  fromAny f = return (import_ f [])
```

Passing functions to foreign code Passing functions the other way, out of Haskell and into our host language, requires slightly more work. While we already had all the pieces of the dynamic import puzzle at our disposal through our earlier implementation of `host`, exports require one more tool in our toolbox: a way to turn a Haskell function into a native host language function.

Much like the `apply` primitive used in the implementation of `host`, the implementation of such an operation is specific to the host language in question. Moreover, as we are dealing with whatever format our chosen compiler has opted to represent functions by, this operation is also dependent on the compiler.

In order to implement this operation, we assume the existence of another function `hfsun_to_host`, to convert a Haskell function f from n `HostAny` arguments to a `HostAny` return value r in the IO monad into a host language function which, when applied to n host language arguments, calls f with those same arguments and returns the r returned by f.

```
foreign import ccall hsfun_to_host :: (HostAny → ... → HostAny) → HostFun
```

But how can we make this operation type check? As we are bound to the types the vanilla foreign function interface lets us marshal, we have no way of applying this function to a variadic Haskell function over `HostAny`s.

We know that, operationally, `hsfun_to_host` expects a Haskell function as its input, but the types do not agree; we must somehow find a way to pass arbitrary data unchanged to our host language. Fortunately, standard Haskell provides us with a way to do exactly what we want: `StablePointer`s [48]. Note that, depending on the Haskell compiler in use, this use of stable pointers may introduce a space leak. This is discussed further in section 6.2, and an alternative solution is presented.

```
import Foreign.StablePtr
import System.IO.Unsafe

foreign import ccall hsfun_to_host' :: StablePtr a → HostFun

hsfun_to_host :: Exportable f ⇒ f → IO HostFun
hsfun_to_host f = hsfun_to_host' `fmap` newStablePtr (mkHostFun f)
```

Just being able to pass Haskell functions verbatim to the host language is not enough. The functions will expect Haskell values as their arguments and return other Haskell values; we need to somehow modify these functions to automatically marshal those arguments and return values. Essentially, we want to map `fromAny` over all input arguments to a function, and `toAny` over its return values. While superficially similar to the implementation of the `Import` class in section 2.3, this task is slightly trickier: where `import_` modifies

an arbitrary number of arguments and performs some action with respect to a monomorphic value – the `HostFun` representation of a host language function – we now need to do the same to a variadic function.

Modifying variadic functions using type families A straightforward application of the `printf` trick used to implement `Import` is not flexible enough to tackle this problem. Instead, we bring in yet another language extension, closed type families [18], to lend us the type level flexibility we need. We begin by defining the `Exportable` type class first encountered in the type signature of `hsfun_to_host`, and a closed type family describing the type level behavior of our function marshalling.

```
type family Host a where
  Host (a → b) = HostAny → Host b
  Host (IO a)  = IO HostAny

class Exportable f where
  mkHostFun :: f → Host f
```

This is relatively straightforward. Inspecting the `Host` type family, we see that applying `mkHostFun` to any eligible function must result in a corresponding function of the same arity – hence the recursive type family instance for `a → b` – but with its arguments and return value replaced by `HostAny`.

Giving the relevant `Exportable` instances is now mostly a matter of making the types match up, and concocting a `ToAny` instance is only a matter of composing our building blocks together.

```
instance ToAny a ⇒ Exportable (IO a) where
  mkHostFun = fmap toAny

instance (FromAny a, Exportable b) ⇒ Exportable (a → b) where
  mkHostFun f = mkHostFun . f . unsafePerformIO . fromAny

instance Exportable f ⇒ ToAny f where
  {-# NOINLINE toAny #-}
  toAny = unsafePerformIO . hsfun_to_host
```

The one interesting instance here is that of the inductive case, where we use `fromAny` in conjunction with `unsafePerformIO` to marshal a single function argument. While using `fromAny` outside the `IO` monad is unsafe in the general case as explained in section 2, this particular instance is completely safe, provided that `mkHostFun` is *not* exported to the user, but only used to implement the `ToAny` instance for functions.

When a function is marshalled into a `HostAny` value and subsequently applied, `fromAny` will be applied unsafely to each of the marshalled function's arguments. There are two cases when this can happen: either the marshalled

function is called from the host language, or it is marshalled back into Haskell and then applied. In the former case, the time of the call is trivially well defined assuming that our target language is not lazy by default. In the latter case, the time of the call is still well defined, as our interface only admits importing functions in the `IO` monad.

Slightly more troubling is the use of `unsafePerformIO` in conjunction with `hsfun_to_host`. According to Reid [48], the creation of stable pointers residing in the `IO` monad – the reason for `hsfun_to_host` residing there as well – is to avoid accidentally duplicating the allocation of the stable pointer, something we can avoid by telling the compiler never to inline the function, ever.

It is also worth pointing out that the concern over duplicating this allocation is only valid where the implementation also has the aforementioned space leak problem, in which case the alternative implementation given in section 6.2 should be preferred anyway.

Marshalling pure functions The above implementation only allows us to pass functions in the `IO` monad to foreign code, but we would also like to support passing pure functions. There are two main obstacles to this:

- The `hsfun_to_host'` function expects a function in the `IO` monad.
- Instantiating `Exportable` for any type `ToAny t ⇒ t` would accidentally add a `ToAny` instance for *any type at all*. Even worse, this instance would be completely bogus for most types, always treating the argument to its `toAny` implementation as a function to be converted into a host language function!

We sidestep the first problem by assuming that `hsfun_to_host'` can determine dynamically whether a function is pure or wrapped in the `IO` monad, and take action accordingly. Another, slightly more verbose, possibility would be to alter the implementation of our marshalling code to use either `hsfun_to_host'` or a function performing the same conversion on pure functions, depending on the type of function being marshalled.

Looking closer at the problematic `ToAny` instance, we find that the `Exportable t ⇒ ToAny t` instance provides `ToAny` for any `Exportable` type, and the `ToAny t ⇒ Exportable t` instance provides `Exportable` in return, creating a loop which creates instances for both type classes matching any type.

The `ToAny t ⇒ Exportable t` instance is necessary for our type level recursion to work out when marshalling pure functions, but we can prevent this instance from leaking to `ToAny` where it would be unreasonably broad by replacing our `ToAny` function instance with two slightly more specific ones.

Figure 11 gives our final implementation of dynamic function exports. Looking at this code we also see why the use of closed type families are

```
import Foreign.StablePtr
import System.IO.Unsafe

foreign import ccall hsfun_to_host' :: StablePtr a → HostFun

hsfun_to_host :: Exportable f ⇒ f → IO HostFun
hsfun_to_host f = hsfun_to_host' `fmap` newStablePtr (mkHostFun f)

type family Host a where
  Host (a → b) = HostAny → Host b
  Host (IO a)  = IO HostAny
  Host a       = HostAny

instance (ToAny a, Host a ~ HostAny) ⇒ Exportable a where
  mkHostFun = toAny

instance (FromAny a, Exportable b) ⇒ ToAny (a → b) where
  {-# NOINLINE toAny #-}
  toAny = unsafePerformIO . hsfun_to_host

instance ToAny a ⇒ ToAny (IO a) where
  {-# NOINLINE toAny #-}
  toAny = unsafePerformIO . hsfun_to_host
```

Figure 11: Dynamic function exports implemented on top of our interface

necessary: the open type families originally introduced by Chakravarty et al [10] do not admit the overlapping type equations required to make pure functions an instance of `Exportable`.

4.2 Static function exports

Very rarely are users prepared to abandon person-decades of legacy code; to reach these users, the ability to expose Haskell functionality to the host language is important. Alas, being implemented as a library, our interface is not capable of `foreign export` declarations. We can, however, implement a substitute on top of it.

Rather than a writing a library which when compiled produces a shared library for consumption by a linker, we give the user access to a function `export` which when executed stores an exported function in a known location, where foreign language code can then access it. While this may seem like a silly workaround, this is how JavaScript programs commonly "link against" third party libraries.

Using the function marshalling implemented in section 4.1, implementing `export` becomes a mere matter of passing a function to the host language,

which then arranges for the function to be available in a known, appropriate location.

```
export :: Exportable f ⇒ String → f → IO ()
export = host "(name, f) ⇒ window['haskell'][name] = f;"
```

4.3 Generic marshalling

Returning to our motivating example with figure 6, we note a conspicuous absence: the `UTCTime` instance of `FromAny` is not defined, yet it is still used by the `host` function in the definition of `getCurrentTime`. Although the instance can be defined in a single line of code, it would still be nice if we could avoid the tedium of writing that one line altogether. Thanks to generic programming and default type class instances, we can.

Our implementation of generic marshalling uses GHC generics [32] and associated language extensions – most notably type operators and scoped type variables – making it specific to GHC-based compilers such as Haste and GHCJS [39]. GHC generics allow us to traverse values of any type as though the type was uniformly defined as a tree of sums, products, constants and metadata, such as record selectors or constructor names.

For the sake of brevity, and as the actual syntax of GHC generics is relatively uninformative due to its generality, we only give the basic method of our implementation is this paper, and only consider the case of marshalling Haskell values into their host language counterparts. Marshalling in the other direction uses the same basic method, and the complete implementation is available from [21] as part of the Haste development suite.

We begin by defining a the data type to represent a host language value while it is being constructed. A value can be either a singleton, a list of values or a dictionary.

```
data Value
  = One HostAny
  | List [HostAny]
  | Dict [(HostString, HostAny)]
```

We then informally define the behavior of our generic marshalling function `gToAny :: Rep a → Value` as follows, where `Rep` is a type provided by `GHC.Generics` to enable generic traversal of its type argument.

- When we reach a *constructor argument* `x` of a type `t` with a `ToAny` instance, we use that instance to marshal `x` and return it as a single value: `One (toAny x)`.
- When we reach a *record selector* metadata node with a selector name `n` and a child node `c`, we recursively marshal `c` and return it paired with

its selector name: `Tree [(n, toAny x)]`.

- When we reach a *constructor* metadata node with a constructor name name `n` and a child node `c`, we recursively marshal `c` and call the resulting value `c'`.

  - If `c'` is a *dictionary*, we add an entry to it to mark the value's constructor name and return the resulting dictionary: `Tree (("tag", toAny n) : c')`.
  - If `c'` is an *empty list*, we simply return the constructor name: `One (toAny n)`.
  - If `c'` is a nonempty *list* or a *single item*, we return a new dictionary consisting of the constructor tag and the `HostAny` encoding of `c'`: `Tree [("tag", toAny n), ("data", toAny c')]`.

- When we reach a *product* node with child nodes `c1` and `c2`, signifying the union of two or more constructor arguments, we recursively marshal `c1` and `c2` into `c1'` and `c2'` respectively. We then *merge* `c1'` and `c2'` and return the result:

```
merge c1' c2'
  where
    merge (One a) (One b)   = List [a, b]
    merge (List a) (One b)  = List (a ++ [b])
    merge (One a) (List b)  = List (a : b)
    merge (List a) (List b) = List (a ++ b)
    merge (Tree a) (Tree b) = Tree (a ++ b)
```

  Note that the case where a tree is merged with a non-tree is undefined. Trees arise *only* from a use of record selectors. Haskell only allows data constructors where either *all* arguments have selectors, or *none* has, meaning that trees and non-trees will never appear in the same product node.

- When we reach a *sum* node, signifying one of several data constructors of a type, we will either have a *left* child or a *right* child. We simply recurse down through whichever child node we have and return the result.

Using this implementation, all that remains is to add a default instance to the `ToAny` class.

```
class ToAny a where
  toAny :: a → HostAny
  default toAny :: (GToAny (Rep a), Generic a) ⇒ a → HostAny
  toAny x =
    case gToAny (from x) of
      One x   → x
      List xs → toAny xs
      Tree d  → mkDict d
```

5 Performance

While increased performance is not a major motivation for this work, it is still important to ascertain that using our library does not entail a major performance hit. To determine the runtime performance of our interface vis a vis the vanilla FFI - a useful baseline for performance comparisons - we have benchmarked a reference implementation of our interface against the vanilla FFI, both implemented for the Haste compiler.

While benchmarking code outside the context of any particular application is often tricky and not necessarily indicative of whole system performance, we hope to give a general idea of how our library fares performance-wise in several different scenarios. To this end, several microbenchmarks were devised:

- *Outbound*, which applies a foreign function to several arguments of type `Double`. The function's return value is discarded, in order to only measure outbound marshalling overhead for primitive types.

- *In-out*, which applies a foreign function to several `Double` arguments and marshals its return value, also of type `Double`, back into Haskell land. This measures inbound as well as outbound marshalling of primitive types.

- *Product types*, which benchmarks the implementation of `getCurrentTime` given in figure 6 against the equivalent implementation given in figure 5, both modified to accept an `UTCTime` value as input in addition to returning the current time, in order to measure outbound marshalling of product types as well as inbound.

- *Higher order import*, which calls a higher order function f using both the vanilla FFI and our method, with a function over a single `Double` value as its argument. The only purpose of f is to call its argument repeatedly, evaluating the speed with which a higher order Haskell

function may be called from external code in addition to the speed of marshalling itself.

These functions were then repeatedly applied in two different contexts: one tight, strict, tail recursive loop, intended to produce as efficient code as possible; and one which simply consists of running `mapM_` over a list of appropriate length, to obtain higher level code which is harder to optimize and analyse for strictness.

The resulting programs, compiled with the Haste compiler which incorporates all the optimizations described in section 3, were then repeatedly executed using the Node.js JavaScript interpreter, and the average run times of the programs using our interface compared against the average run times of their FFI counterparts.

While this may not be the most rigorous of performance evaluations, the results are repeatable, and the methodology is enough for our purposes: getting a rough picture of how much speed we are giving up for a more convenient interface.

The results for each benchmark are given in table 3.1 as the ratio of the run time for our library over the run time for the vanilla FFI.

Outbound Looking at the performance numbers, our library performs surprisingly well in a highly optimized loop, showing no additional overhead over the vanilla FFI. In a less optimizable loop, our interface fares slightly worse. Due to our interface being implemented mainly as a pure Haskell library, the compiler is noticeably worse at figuring out the strictness properties of the program compiled using our library than with the program using the vanilla FFI, leading to some unnecessary creation and evaluation of thunks.

In-out Moving on to the benchmarks where we actually marshal incoming data, the picture is much the same as for the *outbound* benchmark, with a slightly lower performance penalty for the `mapM_` case. This is likely attributable to the extra overhead of storing and evaluating the inbound values for both versions.

Product types Our interface shows a distinct performance advantage when it comes to marshalling more complex values, being as much as *fourteen times* faster in the case of the highly optimized loop. Our assumption about `peek`ing and `poke`ing at pointers in an environment where such low level constructs need to be emulated rather than efficiently implemented on the machine level seems to have been correct.

	Tight loop	`mapM_`
Outbound	1.00	1.16
In-out	1.00	1.05
Product types	0.07	0.37
Higher order import	0.85	0.87

Table 3.1: Execution times compared to the vanilla FFI

This performance advantage is still present in the less optimized loop, although not quite as extreme. Inspecting the generated code, we see that this is caused by the loop in question having a much larger overhead - an indicator that the function calling overheads in both cases are relatively minor compared to other overheads present in the generated code.

Higher order import Our interface seems to compare favorably to the vanilla FFI for this case. The reason for this is not immediately obvious: the two interfaces generate nearly identical code, the vanilla FFI code being slightly more concise. The slight performance difference turns out to be entirely implementation specific, however: the code generated for the vanilla FFI benchmark in some places pass numeric literals around where the code generated for our interface instead passes constant references to the same values. Eliminating this difference by modifying the generated code by hand, the performance difference between the two becomes negligible.

Performance verdict: acceptable Judging by these numbers the performance of our library is quite acceptable, ranging from significantly faster than the vanilla FFI to at most about 15% slower. It is encouraging that our interface's intended use case - marshalling more complex type - is showing tangible performance benefits in addition to the added convenience it affords the user. For code which has no choice but to make a large number of calls to low level host language functions over primitive types in performance critical loops, using the vanilla FFI instead may be an attractive option to reduce the performance penalty - however slight - incurred by our interface, allowing the user to have the FFI cookie and eat it at the same time.

The benchmarks used here are available online from our repository at https://github.com/valderman/ffi-paper.

6 Limitations and discussion

While two of the tree main limitations our interface places on its host language — the presence of a dynamic code evaluation construct and support for first class functions — have hopefully been adequately explained, and their severity slightly alleviated, in sections 2 and 3.3, there are still several design choices and lingering limitations that may need further justification.

6.1 fromAny type level expressiveness

The `fromAny` function used to implement marshalling in section 2 is by definition not total. As its purpose is to convert values of god-knows-what host language type into properly typed Haskell values, from the simplest atomic values to the most complex data structures, the possibility for failure is apparent. Why, then, does its type not admit the possibility of failure, for instance by wrapping the converted value in a `Maybe` or `Either`?

Recall that `fromAny` will almost always be called when automatically converting arguments to and return values from callbacks and imported foreign functions respectively. In this context, even if a conversion were to fail with a `Left "Bad conversion"` error, there is no way for this error value to ever reach the user. The only sensible action for the foreign call to take when encountering an error value would be to throw an exception, informing the user "out of band" rather than by somehow threading an error value to the entire call.

It is then simpler, as well as reducing the amount of error checking overhead necessary, to trust that the foreign code in question is usually well behaved and throw the previously mentioned exception immediately on conversion failure rather than taking a detour via error values, should this trust prove to be misplaced.

6.2 Limitation to garbage collected host languages

The observant reader may notice that up until this point, we have completely ignored something which very much concerns traditional foreign function interfaces: ownership and eventual deallocation of memory.

Our high level interface depends quite heavily on its target language being garbage collected, as having to manually manage memory introduces significant boilerplate code and complexity: the very things this interface aims to avoid. As target platforms *with* garbage collections having to deal with low level details such as memory management is the core motivation for this work, rectifying this "problem" does not fall within the scope of this paper.

Even so, memory management does rear its ugly head in section 4.1, where stable pointers are used to pass data unchanged from Haskell into our host language, and promptly ignored: note the complete absence of calls to `freeStablePtr`. Implementing our interface for the Haste compiler, this is not an issue: Haste makes full use of JavaScript's garbage collection capabilities to turn stable pointers into fully garbage collected aliases of the objects pointed to. It is, however, quite conceivable for an implementation to perform some manual housekeeping of stable pointers even in a garbage collected language, in which case this use of our interface will cause a space leak as nobody is keeping track of all the stable pointers we create.

As the stable pointers in question are never dereferenced or otherwise used within Haskell, this hypothetical space leak can be eliminated by replacing stable pointers with a slight bit of unsafe, implementation-specific magic.

```
import Unsafe.Coerce
import Foreign.StablePtr hiding (newStablePtr)

data FakeStablePtr a
fakeStablePtr :: a → FakeStablePtr a

newStablePtr :: a → StablePtr
newStablePtr = unsafeCoerce . fakeStablePtr
```

The `FakeStablePtr` type and the function by the same name are used to mimic the underlying structure of `StablePtr`. This makes its exact implementation specific to the Haskell compiler in question, unlike the "proper" solution based on actual stable pointers. The Haste compiler, being based on GHC, has a very straightforward representation for stable pointers, merely wrapping the "machine" level pointer in a single layer of indirection, giving us the following implementation of fake stable pointers:

```
data FakeStablePtr a = Fake !a

fakeStablePtr = Fake
```

Thus, we may choose our implementation strategy depending on the capabilities of our target compiler. For a single implementation targeting multiple platforms however, proper stable pointers are the safer solution.

6.3 Restricting imports to the IO monad

The interface presented in this paper does not support importing pure functions; any function originating in the host language must be safely locked up within the IO monad. This may be seen as quite a drawback, as a host language function operating solely over local state is definitely

not beyond the realms of possibility. Looking at our implementation of function exports for pure functions, it seems that it would be possible to implement imports in a similar way, and indeed we could.

However, "could" is not necessarily isomorphic to "should". Foreign functions do, after all, come from the unregulated, disorderly world outside the confines of the type checker. Haskell's type system does not allow us to mix pure functions with possibly impure ones, and for good reason. It is not clear that we should lift this restriction just because a function is defined in another language.

Moreover, as explained in section 2, marshalling inbound data is in many cases an inherently effectful operation, particularly when involving complex data structures. Permitting the import of pure functions, knowing fully well that a race condition exists in the time window between the import's application and the resulting thunk's evaluation, does not strike us as a shining example of safe API design.

Better, then, to let the user import their foreign code in the IO monad and explicitly vouch for its purity, using `unsafePerformIO` to bring it into the world of pure functions.

6.4 Blocking in non-concurrent environments

A particularly neat feature of the foreign function interface employed by the GHCJS compiler is the ability for foreign host code to suspend execution while waiting for an event to occur, even though its JavaScript host environment is devoid of any concurrency support [39]. This is accomplished by giving imported functions an extra parameter: a continuation to be called upon completion of the foreign operation to resume execution of the Haskell program, instead of simply returning like a "normal" JavaScript function would.

This functionality is not supported by our interface. GHCJS accomplishes this by outputting continuation passing code which is executed by a clever trampolining machinery. Supporting this feature would tie the interface to a particular code generation strategy as well as add considerable complexity; a price we deem too high to pay for this feature.

Instead, this functionality can be implemented on top of our interface without much difficulty using a construct dubbed the "poor man's concurrency monad" [12]; a monad implementing coarse-grained, preemptive multitasking with blocking synchronization variables in non-concurrent environments.

6.5 Summary

In this chapter we have presented the design and implementation of the novel, lightweight *Haste.Foreign* foreign function interface. Haste.Foreign allows boilerplate-free interoperation with native JavaScript code, while simultaneously allowing the programmer a large degree of control over the marshalling process when desired. We have discussed the various limitations imposed by the interface and contrasted it with the foreign function interfaces of competing compilers, and given performance enhancements and extensions to alleviate most of said limitations.

We have shown that despite enabling automatic marshalling of complex types such as higher order functions and algebraic data types, the performance of Haste.Foreign is at least on par with that of Haskell's vanilla foreign function interface when implemented for the Haste compiler.

Haste.Foreign requires no compiler modifications, but can be implemented entirely as a library and is portable across Haskell dialects as well as high level target environments fulfilling certain criteria.

Bridging the client-server gap

This chapter is based on the paper *A Seamless, Client-centric Programming Model for Type Safe Web Applications* [22], presented at Haskell Symposium '14; a joint work with my supervisor Koen Claessen.

It describes the design and implementation of *Haste.App*, a programming model for type safe, distributed web applications. Section 1 introduces several problems with contemporary industry practices in the area. Section 2 proposes a different approach to developing distributed web applications, and section 3 gives a reference implementation of this model. In section 4.1 we discuss limitations of and alternatives to our approach, and argue the efficiacy of the Haste.App programming model.

1 Background

Most web applications – traditional ones as well as modern, rich client single page applications – are intended to facilitate communication, data storage or some other task involving a centralized resource. This makes a significant server component, in addition to the client code running in the user's web browser, a major part of the application. This server component is usually implemented as a completely separate program, and communicates with the client program over some network protocol.

This state of things is not a conscious design choice - most web applications are conceptually a single entity, not two programs which just happen to talk to each other over a network - but a consequence of there being a large, distributed network between the client and server parts. However, such implementation details should not be allowed to dictate the way we structure and reason about our applications - clearly, an abstraction is called for.

For a more concrete example, let's say that we want to implement a simple "chatbox" component for a website, to allow visitors to discuss the site's content in real time. Using mainstream development practices and recent technologies such as WebSockets [31], we may come up with

```
function handshake(sock) {sock.send('helo');}
function chat(sock, msg) {sock.send('text' + msg);}

window.onload = function() {
  var logbox = document.getElementById('log');
  var msgbox = document.getElementById('message');
  var sock = new WebSocket('ws://example.com');

  sock.onmessage = function(e) {
    logbox.value = e.data + LINE + logbox.value;
  };

  sock.onopen = function(e) {
    handshake(sock);
    msgbox.addEventListener('keydown', function(e) {
      if(e.keyCode == 13) {
        var msg = msgbox.value;
        msgbox.value = '';
        chat(sock, msg);
      }
    });
  };
};
```

Figure 12: JavaScript chatbox implementation

something like the program in figure 12 for our client program. In addition, a corresponding server program would need to be written to handle distribution of messages among clients. We will not give such an implementation here, as we do not believe it necessary to state the problem at hand.

Since the "chatbox" application is very simple - users should only be able to send and receive text messages in real time - we opt for a very simple design. Two UI elements, `logbox` and `msgbox`, represent the chat log and the text area where the user inputs their messages respectively. When a message arrives, it is prepended to the chat log, making the most recent message appear at the top of the log window, and when the user hits the return key in the input text box the message contained therein is sent and the input text box is cleared.

Messages are transmitted as strings, with the initial four characters indicating the type of the message and the rest being the optional payload. There are only two messages: a handshake indicating that a user wants to join the conversation, and a broadcast message which sends a line of text to all connected users via the server. The only messages received from the server are new chat messages, delivered as simple strings.

This code looks solid enough by web standards, but even this simple

piece of code contains no less than three asynchronous callbacks, two of which both read and modify the application's global state. This makes the program flow non-obvious, and introduces unnecessary risk and complexity through the haphazard state modifications.

Moreover, this code is not very extensible. If this simple application is to be enhanced with new features down the road, the network protocol will clearly need to be redesigned. However, if we were developing this application for a client, said client would likely not want to pay the added cost for the design and implementation of features she did not - and perhaps never will - ask for.

Should the protocol need updating in the future, how much time will we need to spend on ensuring that the protocol is used properly across our entire program, and how much extra work will it take to keep the client and server in sync? How much code will need to be written twice, once for the client and once for the server, due to the unfortunate fact that the two parts are implemented as separate programs, possibly in separate languages?

Above all, is it really necessary for such a simple program to involve client/server architectures and network protocol design at all?

2 A seamless programming model

There are many conceivable improvements to the mainstream web development model described in the previous section. We propose an alternative programming model based on Haskell, in which web applications are written as a single program rather than as two independent parts that just so happen to talk to each other.

Instead we propose a programming model, dubbed "Haste.App", with the following properties:

- The programming model is synchronous, giving the programmer a simple, linear view of the program flow, eliminating the need to program with callbacks and continuations.

- Side-effecting code is explicitly designated to run on either the client or the server using the type system while pure code can be shared by both. Additionally, general IO computations may be lifted into both client and server code, allowing for safe IO code reuse within the confines of the client or server designated functions.

- Client-server network communication is handled through statically typed RPC function calls, extending the reach of Haskell's type

checker over the network and giving the programmer advance warning when she uses network services incorrectly or forgets to update communication code as the application's internal protocol changes.

- Our model takes the view that the client side is the main driver when developing web applications and accordingly assigns the server the role of a computational and/or storage resource, tasked with servicing client requests rather than driving the program. While it is entirely possible to implement a server-to-client communication channel on top of our model, we believe that choosing one side of the heterogeneous client-server relation as the master helps keeping the program flow linear and predictable.
- The implementation is built as a library on top of the GHC and Haste Haskell compilers, requiring little to no specialized compiler support. Programs are compiled twice; once with Haste and once with GHC, to produce the final client and server side code respectively.

2.1 A first example

While explaining the properties of our solution is all well and good, nothing compares to a good old Hello World example to convey the idea. We begin by implementing a function which prints a greeting to the server's console.

```
import Haste.App

helloServer :: String → Server ()
helloServer name =
  liftIO $ putStrLn (name ++ " says hello!")
```

Computations exclusive to the server side live in the `Server` monad. This is basically an IO monad, as can be seen from the regular `putStrLn IO` computation being lifted into it, with a few extra operations for session handling; its main purpose is to prevent the programmer from accidentally attempting to perform client-exclusive operations, such as popping up a browser dialog box, on the server.

Next, we need to make the `helloServer` function available as an RPC function and call it from the client.

```
main :: App Done
main = do
  greetings ← remote helloServer

  runClient $ do
    name ← prompt "Hi there, what is your name?"
    onServer (greetings <.> name)
```

The `main` function is, as usual, the entry point of our application. In contrast to traditional applications which live either on the client or on the server and begin in the `IO` monad, Haste.App applications live on both and begin execution in the `App` monad which provides some crucial tools to facilitate typed communication between the two.

The `remote` function takes an arbitrary function, provided that all its arguments as well as its return value are serializable through the `Serialize` type class, and produces a typed identifier which may be used to refer to the remote function. In this example, the type of `greetings` is `Remote (String → Server ())`, indicating that the identifier refers to a remote function with a single `String` argument and no return value. Remote functions all live in the `Server` monad. The part of the program contained within the `App` monad is executed on both the server and the client, albeit with slightly different side effects, as described in section 3.

After the `remote` call, we enter the domain of client-exclusive code with the application of `runClient`. This function executes computations in the `Client` monad which is essentially an IO monad with cooperative multitasking added on top, to mitigate the fact that JavaScript has no native concurrency support. `runClient` does not return, and is the only function with a return type of `App Done`, which ensures that each `App` computation contains exactly one client computation.

In order to make an RPC call using an identifier obtained from `remote`, we must supply it with an argument. This is done using the `<.>` operator. It might be interesting to note that its type, `Serialize a ⇒ Remote (a → b) → a → Remote b`, is very similar to the type of the `<*>` operator over applicative functors. This is not a coincidence; `<.>` performs the same role for the `Remote` type as `<*>` performs for applicative functors. The reason for using a separate operator for this instead of making `Remote` an instance of `Applicative` is that since functions embedded in the `Remote` type exist only to be called over a network, such functions must only be applied to arguments which can be serialized and sent over a network connection. When a `Remote` function is applied to an argument using `<.>`, the argument is serialized and stored inside the resulting `Remote` object, awaiting dispatch. `Remote` computations can thus be seen as explicit representations of closures.

After applying the value obtained from the user to the remote function, we apply the `onServer` function to the result, which dispatches the RPC call to the server. `onServer` will then block until the RPC call returns.

To run this example, an address and a port must be provided so that the client knows which server to contact. There are several ways of doing this: using the GHC plugin system, through Template Haskell or by slightly

altering how program entry points are treated in a compiler or wrapper script, to name a few. A non-intrusive method when using the GHC/Haste compiler pair would be to add `-main-is setup` to both compilers' command line and add the `setup` function to the source code.

```
setup :: IO ()
setup =
  runApp (mkConfig "ws://localhost:1111" 1111) main
```

This will instruct the server binary to listen on the port 1111 when started, and the client to attempt contact with that port on the local machine. The exact mechanism chosen to provide the host and port are implementation specific, and will in the interest of brevity not be discussed further.

2.2 Using server side state

While the Hello Server example illustrates how client-server communication is handled, most web applications need to keep some server side state as well. How can we create state holding elements for the server which are not accessible to the client?

To accomplish this, we need to introduce a way to lift arbitrary IO computations, but ensure that said computations are executed on the server and nowhere else. This is accomplished using a more restricted version of `liftIO`:

```
liftServerIO :: IO a → App (Server a)
```

`liftServerIO` performs its argument computation once on the server, in the `App` monad, and then returns the result of said computation inside the `Server` monad so that it is only reachable by server side code. Any client side code is thus free to completely ignore executing computations lifted using `liftServerIO`; since the result of a server lifted computation is never observable on the client, the client has no obligation to even produce such a value. Figure 13 shows how to make proper use of server side state.

2.3 The chatbox, revisited

Now that we have seen how to both implement network communication and work with server side state, we are ready to revisit the chatbox program from section 1, this time using our improved programming model. Since we are now writing the entire application, both client and server, as opposed to the client part from our motivating example, our program has three new responsibilities.

- We need to add connecting users to a list of message recipients;

```
main = do
  remoteref ← liftServerIO $ newIORef 0

  count ← remote $ do
    r ← remoteref
    liftIO $ atomicModifyIORef r (λv → (v+1, v+1))

  runClient $ do
    visitors ← onServer count
    alert ("Your are visitor #" ++ show visitors)
```

Figure 13: server side state: doing it properly

- users leaving the site need to be removed from the recipient list; and
- chat messages need to be distributed to all users in the list.

With this in mind, we begin by importing a few modules we are going to need and define the type for our recipient list.

```
import Haste.App
import Haste.App.Concurrent
import qualified Control.Concurrent as CC

type Recipient = (SessionID, CC.Chan String)
type RcptList = CC.MVar [Recipient]
```

We use an `MVar` from `Control.Concurrent` to store the list of recipients. A recipient will be represented by a `SessionID`, an identifier used by Haste.App to identify user sessions, and an `MVar` into which new chat messages sent to the recipient will be written as they arrive. Next, we define our handshake RPC function.

```
srvHello :: Server RcptList → Server ()
srvHello remoteRcpts = do
  recipients ← remoteRcpts
  sid ← getSessionID
  liftIO $ do
    rcptChan ← CC.newChan
    CC.modifyMVar recipients $ λcs →
      return ((sid, rcptChan):cs, ())
```

An `MVar` is associated with the connecting client's session identifier, and the pair is prepended to the recipient list. Notice how the application's server state is passed in as the function's argument, wrapped in the `Server` monad in order to prevent client-side inspection.

```
srvSend :: Server RcptList → String → Server ()
srvSend remoteRcpts message = do
    rcpts ← remoteRcpts
    liftIO $ do
      recipients ← CC.readMVar rcpts
      mapM_ (flip CC.writeChan message) recipients
```

The send function is slightly more complex. The incoming message is written to the `Chan` corresponding to each active session.

```
srvAwait :: Server RcptList → Server String
srvAwait remoteRcpts = do
  rcpts ← remoteRcpts
  sid ← getSessionID
  liftIO $ do
    recipients ← CC.readMVar rcpts
    case lookup sid recipients of
      Just mv → CC.readChan mv
      _       → fail "Unregistered session!"
```

The final server operation, notifying users of pending messages, finds the appropriate `Chan` to wait on by searching the recipient list for the session identifier of the calling user, and then blocks until a message arrives in said `MVar`. This is a little different from the other two operations, which perform their work as quickly as possible and then return immediately.

If the caller's session identifier could not be found in the recipient list, it has for some reason not completed its handshake with the server. If this is the case, we simply drop the session by throwing an error; an exception will be thrown to the client. No server side state needs to be cleaned up as the very lack of such state was our reason for dropping the session.

Having implemented our three server operations, all that's left is to tie them to the client. In this tying, we see our main advantage over the JavaScript version in section 1 in action: the `remote` function builds a strongly typed bridge between the client and the server, ensuring that any future enhancements to our chatbox program are made safely, in one place, instead of being spread about throughout two disjoint code bases.

```
main :: App Done
main = do
  recipients ← liftServerIO $ CC.newMVar []

  hello ← remote $ srvHello recipients
  awaitMsg ← remote $ srvAwait recipients
  sendMsg ← remote $ srvSend recipients

  runClient $ do
    withElems ["log","message"] $ λ[log,msgbox] → do
      onServer hello
```

Notice that the `recipients` list is passed to our three server operations *before* they are imported; since `recipients` is a mutable reference created on the server and inaccessible to client code, it is not possible to pass it over the network as an RPC argument. Even if it were possible, passing server-private state back and forth over the network would be quite inappropriate due to privacy and security concerns.

The `withElems` function is part of the Haste compiler's bundled DOM manipulation library; it locates references to the DOM nodes with the given identifiers and passes said references to a function. In this case the variable `log` will be bound to the node with the identifier "log", and `msgbox` will be bound to the node identified by "message". These are the same DOM nodes that were referenced in our original example, and refer to the chat log window and the text input field respectively. After locating all the needed UI elements, the client proceeds to register itself with the server's recipient list using the `hello` remote computation.

```
let recvLoop chatlines = do
      setProp log "value" $ unlines chatlines
      message ← onServer awaitMsg
      recvLoop (message : chatlines)
fork $ recvLoop []
```

The `recvLoop` function perpetually asks the server for new messages and updates the chat log whenever one arrives. Note that unlike the `onmessage` callback of the JavaScript version of this example, `recvLoop` is acting as a completely self-contained process with linear program flow, keeping track of its own state and only reaching out to the outside world to write its state to the chat log whenever necessary. As the `awaitMsg` function blocks until a message arrives, `recvLoop` will make exactly one iteration per received message.

```
runClient    :: Client () → App Done
liftServerIO :: IO a → App (Server a)
remote       :: Remotable a
             ⇒ a → App (Remote a)

onServer     :: Remote (Server a) → Client a
(<.>)        :: Serialize a
             ⇒ Remote (a → b) → a → Remote b

getSessionID :: Server SessionID
```

Figure 14: Types of the Haste.App core functions

```
msgbox `onEvent` OnKeyPress $ λ13 → do
  msg ← getProp msgbox "value"
  setProp msgbox "value" ""
  onServer (sendMsg <.> msg)
```

This is the final part of our program; we set up an event handler to clear the input box and send its contents off to the server whenever the user hits return (character code 13) while the input box has focus.

The discerning reader may be slightly annoyed at the need to extract the contents from `Remote` values at each point of use. Indeed, in a simple example such as this, the source clutter caused by this becomes a disproportionate irritant. Fortunately, most web applications tend to have more complex client-server interactions, reducing this overhead significantly.

A complete listing of the core functions in Haste.App is given in table 4.1, and their types are given in figure 14.

3 Implementation

Our implementation is built in three layers: the compiler layer, the concurrency layer and the communication layer. The concurrency and communication layers are simple Haskell libraries, portable to any other pair of standard Haskell compilers with minimal effort.

To pass data back and forth over the network, messages are serialized using JSON, a fairly lightweight format used by many web applications, and sent using the HTML5 WebSockets API. This choice is completely arbitrary, guided purely by implementation convenience. It is certainly not the most performant choice, but can be trivially replaced with something more suitable as needed.

The implementation described here is a slight simplification of our implementation, removing some performance enhancements and error

Function	Purpose
`runClient`	Lift a single `Client` computation into the `App` monad. Must be at the very end of an `App` computation, which is enforced by the type system.
`liftServerIO`	Lift an IO computation into the `App` monad. The computation and its result are exclusive to the server, as enforced by the type system, and are not observable on the client.
`remote`	Make a server side function available to be called remotely by the client.
`onServer`	Dispatch a remote call to the server and wait for its completion. The result of the remote computation is returned on the client after it completes.
`<.>`	Apply a `remote` function to a serializable argument.
`getSessionID`	Get the unique identifier for the current session. This is a pure convenience function, to relieve programmers of the burden of session bookkeeping.

Table 4.1: Core functions of Haste.App

handling clutter in the interest of clarity. The complete implementation is available for download as part of the Haste development suite.

Two compilers The principal trick to our solution is compiling the same program twice; once with a compiler that generates the server binary, and once with one that generates JavaScript. Conditional compilation is used for a select few functions, to enable slightly different behavior on the client and on the server as necessary. Using Haskell as the base language of our solution leads us to choose GHC as our server side compiler by default. We chose the Haste compiler to provide the client side code, mainly owing to our great familiarity with it and its handy ability to make use of vanilla Haskell packages from Hackage.

The `App` monad The `App` monad is where remote functions are declared, server state is initialized and program flow is handed over to the `Client` monad. Its definition is as follows.

```
type CallID = Int
type Method = [JSON] → IO JSON
type AppState = (CallID, [(CallID, Method)])
newtype App a = App (StateT AppState IO a)
  deriving (Functor, Applicative, Monad)
```

As we can see, `App` is a simple state monad, with underlying IO capabilities to allow server side computations to be forked from within it. Its `CallID` state element contains the identifier to be given to the next remote function, and its other state element contains a mapping from identifiers to remote functions.

What makes `App` interesting is that computations in this monad are executed on both the client and the server; once on server startup, and once in the startup phase of each client. Its operations behave slightly differently depending on whether they are executed on the client or on the server. Execution is deterministic, ensuring that the same sequence of `CallID`s are generated during every run, both on the server and on all clients. This is necessary to ensure that any particular call identifier always refers to the same server side function on all clients.

After all common code has been executed, the program flow diverges between the client and the server; client side, `runClient` launches the application's `Client` computation whereas on the server, this computation is discarded, and the server instead goes into an event loop, waiting for calls from the client.

The workings of the `App` monad basically hinges on the `Server` and `Remote` abstract data types. `Server` is the monad wherein any server side code is

contained, and `Remote` denotes functions which live on the server but can be invoked remotely by the client. The implementation of these types and the functions that operate on them differ between the client and the server.

3.1 Client side implementation

We begin by looking at the client side implementation for those two types.

```
data Server a = ServerDummy
data Remote a = Remote CallID [JSON]
```

The `Server` monad is quite uninteresting to the client; since operations performed within it can not be observed by the client in any way, such computations are simply represented by a dummy value. The `Remote` type contains the identifier of a remote function and a list of the serialized arguments to be passed when invoking it. In essence, it is an explicit representation of a remote closure. Such closures can be applied to values using the `<.>` operator.

```
(<.>) :: Serialize a
      ⇒ Remote (a → b) → a → Remote b
(Remote identifier args) <.> arg =
  Remote identifier (toJSON arg : args)
```

The `remote` function is used to bring server side functions into scope on the client as `Remote` functions. It is implemented using a simple counter which keeps track of how many functions have been imported so far and thus which identifier to assign to the next remote function.

```
remote :: Remotable a ⇒ a → App (Remote a)
remote _ = App $ do
  (next_id, remotes) ← get
  put (next_id+1, remotes)
  return (Remote next_id [])
```

As the remote function lives on the server, the client only needs an identifier to be able to call on it. The remote function is thus ignored, so that it can be optimized out of existence in the client executable. Looking at its type, we can see that `remote` accepts any argument instantiating the `Remotable` class. `Remotable` is defined as follows.

```
class Remotable a where
  mkRemote :: a → ([JSON] → Server JSON)

instance Serialize a ⇒ Remotable (Server a) where
  mkRemote m = λ_ → fmap toJSON m

instance (Serialize a, Remotable b) ⇒
         Remotable (a → b) where
  mkRemote f =
    λ(x:xs) → mkRemote (f $ fromJSON x) xs
```

In essence, any function, over any number of arguments, which returns a serializable value in the `Server` monad can be imported. The `mkRemote` function makes use of a well-known type class trick for creating statically typed variadic functions, and works very much like the `printf` function of Haskell's standard library [1].

The final function operating on these types is `liftServerIO`, used to initialize state holding elements and perform other setup functionality on the server.

```
liftServerIO :: IO a → App (Server a)
liftServerIO _ = App $ return ServerDummy
```

As we can see, the implementation is as simple as can be. Since `Server` is represented by a dummy value on the client, we just return said value.

3.2 Server side implementation

The server side representation of the `Server` and `Remote` types are in a sense the opposites of their client side counterparts.

```
newtype Server a = Server (ReaderT SessionInfo IO a)
  deriving (Functor, Applicative, Monad, MonadIO)
data Remote a = RemoteDummy
```

Where the client is able to do something useful with the `Remote` type but can't touch `Server` values, the server has no way to inspect `Remote` functions, and thus only has a no-op implementation of the `<.>` operator. On the other hand, it does have full access to the values and side effects of the `Server` monad, which is an IO monad with some additional session data for the convenience of server side code.

`Server` values are produced by the `liftServerIO` and `remote` functions. `liftServerIO` is quite simple: the function executes its argument immediately and the result is returned, tucked away within the `Server` monad.

```
liftServerIO :: IO a → App (Server a)
liftServerIO m = App $ do
  x ← liftIO m
  return (return x)
```

The server version of `remote` is a little more complex than its client side counterpart. In addition to keeping track of the identifier of the next remote function, the server side `remote` pairs up remote functions with these identifiers in an identifier-function mapping.

```
remote f = App $ do
  (next_id, remotes) ← get
  put (next_id+1, (next_id, mkRemote f) : remotes)
  return RemoteDummy
```

This concept of client side identifiers being sent to the server and used as indices into a table mapping identifiers to remotely accessible functions is an extension of the concept of "static values" introduced by Epstein et al with Cloud Haskell [23], which is discussed further in section 4.2.

The server side dispatcher After the `App` computation finishes, the identifier-function mapping accumulated in its state is handed over to the server's event loop, where it is used to dispatch the proper functions for incoming calls from the client.

```
onEvent :: [(CallID, Method)] → JSON → IO ()
onEvent mapping incoming = do
  let (nonce, identifier, args) = fromJSON incoming
      Just f = lookup identifier mapping
  result ← f args
  webSocketSend $ toJSON (nonce, result)
```

The function corresponding to the RPC call's identifier is looked up in the identifier-function mapping and applied to the received list of arguments. The return value is paired with a nonce provided by the client to tie it to its corresponding RPC call, since there may be several such calls in progress at the same time. The pair is then sent back to the client.

Note that during normal operation, it is not possible for the client to submit an RPC call with a non-existent call identifier, hence the irrefutable pattern match on `Just f`. Should this pattern match fail, this is a sure sign of malicious tampering; the resulting exception is caught and the session is dropped as it is no longer meaningful to continue.

The `Client` monad and the `onServer` function As synchronous network communication is one of our stated goals, it is clear that we will need some

kind of blocking primitive. Since JavaScript does not support any kind of blocking, we will have to implement this ourselves.

A solution is given in the *poor man's concurrency monad* [12]. Making use of a continuation monad with primitive operations for forking a computation and atomically lifting an IO computation into the monad, it is possible to implement cooperative multitasking on top of the non-concurrent JavaScript runtime. This monad allows us to implement `MVar`s as our blocking primitive, with the same semantics as their regular Haskell counterpart [43]. This concurrency-enhanced IO monad is used as the basis of the `Client` monad.

```
type Nonce = Int
type ClientState = (Nonce, Map Nonce (MVar JSON))
type Client = StateT ClientState Conc
```

Aside from the added concurrency capabilities, the `Client` monad only has a single particularly interesting operation: `onServer`.

```
newResult :: Client (Nonce, MVar JSON)
newResult = do
  (nonce, m) ← get
  var ← liftIO newEmptyMVar
  put (nonce+1, insert nonce var m)
  return (nonce, mv)

onServer :: Serialize a
         ⇒ Remote (Server a) → Client a
onServer (Remote identifier args) = do
  (nonce, var) ← newResult
  webSocketSend $
    toJSON (nonce, identifier, reverse args)
  fromJSON <$> takeMVar var
```

The `createResultMVar` function creates a new `MVar`, paired with its corresponding nonce in the client's map of nonces to *result variables*.

After a call is dispatched, `onServer` blocks, waiting for its result variable to be filled with the result of the call. Filling this variable is the responsibility of the *receive callback*, which is executed every time a message arrives from the server.

```
onMessage :: JSON → Client ()
onMessage response = do
  let (nonce, result) = fromJSON response
  (n, m) ← get
  put (n, delete nonce m)
  putMVar (m ! nonce) result
```

As we can see, the implementation of our programming model is rather simple and requires no bothersome compiler modifications or language

extensions, and is thus easily portable to other Haskell compilers.

4 Limitations and discussion

4.1 Limitations

Client centricity Unlike most related work, our approach takes a firm stand, regarding the client as the driver in the client-server relationship with the server taking on the role of a passive computational or storage resource. The server may thus not call back into the client at arbitrary points but is instead limited to returning answers to client side queries. This is clearly less flexible than the back-and-forth model of Sunroof and Cheerp or the shared variables of Conductance. However, we believe that this restriction makes program flow easier to follow and comprehend. Like the immutability of Haskell, this model gives programmers a not-so-subtle hint as to how they may want to structure their programs. Extending our existing model with an `onClient` counterpart to `onServer` would be a simple task, but we are not quite convinced that there is value in doing so.

Environment consistency As our programming model uses two different compilers to generate client and server code, it is crucial to keep the package environments of the two in sync. A situation where, for instance, a module is visible to one compiler but not to the other will render many programs uncompilable until this inconsistency is fixed.

This kind of divergence can be worked around using conditional compilation, but is highly problematic even so; using a unified package database between the two compilers, while problematic due to the differing natures of native and JavaScript compilation respectively, would be a significant improvement in this area.

4.2 Inspiration and alternatives to `remote`

One crucial aspect of implementing cross-network function calls is the issue of data representation: the client side of things must be able to obtain some representation of any function it may want to call on the server.

In our solution, this representation is obtained through the use of the `remote` function, which when executed on the server pairs a function with a unique identifier, and when executed on the client returns said identifier so that the client may now refer to the function. While this has the advantage of being simple to implement, one major drawback of this method is that all functions must be explicitly imported in the `App` monad prior to being called over the network.

This approach was inspired by Cloud Haskell [23], which introduces the notion of "static values"; values which are known at compile time. Codifying this concept in the type system, to enable it to be used as a basis for remote procedure calls, unfortunately requires some major changes to the compiler. Cloud Haskell has a stopgap measure for unmodified compilers wherein a remote table, pairing values with unique identifiers, is kept. This explicit bookkeeping relies on the programmer to assign appropriate types to both values themselves and their identifiers, breaking type safety.

The astute reader may notice that this is exactly what the `remote` function does as well, the difference being that `remote` links the identifier to the value it represents on the type level, preventing the user from calling non-existent remote functions or breaking the program's type safety in other ways.

Another approach to this problem is defunctionalization [14], a program transformation wherein functions are translated into algebraic data types. This approach would allow the client and server to use the same actual code; rather than passing an identifier around, the client would instead pass the actual defunctionalized code to the server for execution. This would have the added benefit of allowing functions to be arbitrarily composed before being remotely invoked.

This approach also requires significant changes to the compiler, making it unsuitable for our use case. Moreover, we are not entirely convinced about the wisdom of allowing server side execution of what is essentially arbitrary code sent from the client which, in a web application context, is completely untrustworthy. While analyzing code for improper behavior is certainly possible, designing and enforcing a security policy sufficiently strict to ensure correct behavior while flexible enough to be practically useful would be an unwelcome burden on the programmer.

4.3 Advantages of our approach

We believe that our approach has a number of distinct advantages over the related work described in section 2 of chapter 1.

Our approach gives the programmer access to the same strongly typed, general-purpose functional language on both client and server; any code which may be of use to both client and server is effortlessly shared, leading to less duplication of code and increased possibilities for reusing third party libraries.

Interactive multiplayer games are one type of application where this code sharing may have a large impact. In order to ensure that players are not cheating, a game server must keep track of the entire game state and send updates to clients at regular intervals. However, due to network

latency, waiting for server input before rendering each and every frame is completely impractical. Instead, the usual approach is to have each client continuously compute the state of the game to the best of its knowledge, rectifying any divergence from the game's "official" state whenever an update arrives from the server. In this scenario, it is easy to see how reusing much of the same game logic between the client and the server would be very important.

Any and all communication between client and server is both strongly typed and made explicit by the use of the `onServer` function, with the programmer having complete control over the serialization and deserialization of data using the appropriate type classes. Aside from the obvious advantages of type safety, making the crossing of the network boundary explicit aids the programmer in making an informed decision as to when and where server communication is appropriate, as well as helps preventing accidental transmission of sensitive information intended to stay on either side of the network.

Our programming model is implemented as a library, assuming only two Haskell compilers, one targeting JavaScript and one targeting the programmer's server platform of choice. While we use Haste as our JavaScript-targeting compiler, modifying our implementation to use GHCJS or even the JavaScript backend of UHC would be trivial. This implementation not only allows for greater flexibility, but also eliminates the need to modify complex compiler internals.

4.4 Summary

In this chapter we have presented the *Haste.App* programming model for strongly typed, distributed web applications. Although similar programming models exist – the more notable ones are described in section 2 of chapter 1 – to our knowledge Haste.App is the first one to offer the use of a full more or less mainstream programming language, as opposed to a more restricted DSL, with strong guarantees of type safety while being implementable completely without modifying the underlying language.

We have discussed the design of Haste.App, arguing that its restrictions compared to many competing models are actually helpful in enabling programmers to write programs with less defects. We have also given a reference implementation of Haste.App built on standard web technologies. While the reference implementation uses the Haste and GHC compilers, it could equally easily be built on top of another pair of Haskell compilers, such as UHC or GHCJS, owing to its lightweight and general nature.

Bibliography

[1] L. Augustsson and B. Massey. The Text.Printf module. http://hackage.haskell.org/package/base-4.8.1.0/docs/Text-Printf.html, 2013.
→ 2 citations on 2 pages: 59 and 90

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.
→ 1 citation on page: 17

[3] V. Balat, P. Chambart, and G. Henry. Client-server Web applications with Ocsigen. In *WWW2012*, page 59, Lyon, France, Apr. 2012.
→ 2 citations on 2 pages: 2 and 9

[4] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *ACM SIGPLAN Notices*, volume 34, pages 129–140. ACM, 1998.
→ 1 citation on page: 2

[5] J. Bracker and A. Gill. Sunroof: A monadic DSL for generating JavaScript. In M. Flatt and H.-F. Guo, editors, *Practical Aspects of Declarative Languages*, volume 8324 of *Lecture Notes in Computer Science*, pages 65–80. Springer International Publishing, 2014.
→ 1 citation on page: 9

[6] E. Brady. Cross-platform compilers for functional languages. *Under consideration for Trends in Functional Programming*, 2015.
→ 1 citation on page: 6

[7] E. Bruël and J. M. Jansen. Implementing a non-strict purely functional language in JavaScript. *Implementation of Functional Languages*, 2010.
→ 1 citation on page: 4

[8] M. M. Chakravarty. *The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*. 2003.
→ 1 citation on page: 51

[9] M. M. Chakravarty. Foreign inline code: Systems demonstration. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 119–120, New York, NY, USA, 2014. ACM.
→ 1 citation on page: 7

[10] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM.
→ 1 citation on page: 67

[11] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 122–133, New York, NY, USA, 2010. ACM.
→ 1 citation on page: 10

[12] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9:313–323, 1999.
→ 3 citations on 3 pages: 10, 75, and 92

[13] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer Berlin Heidelberg, 2007.
→ 1 citation on page: 10

[14] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM.
→ 1 citation on page: 94

[15] A. Dijkstra, J. Stutterheim, A. Vermeulen, and S. Swierstra. Building JavaScript applications with Haskell. In R. Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 37–52. Springer Berlin Heidelberg, 2013.
→ 3 citations on 3 pages: 2, 7, and 19

[16] C. Done. Fay, JavaScript, etc. http://chrisdone.com/posts/fay, 2012.
→ 1 citation on page: 5

[17] B. Eich. From ASM.js to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/, 2015.
→ 1 citation on page: 15

[18] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, New York, NY, USA, 2014. ACM.
→ 1 citation on page: 65

[19] A. Ekblad. Towards a declarative web. Master's thesis, University of Gothenburg. Also available from http://ekblad.cc/hastereport.pdf.
→ 2 citations on 2 pages: 2 and 20

[20] A. Ekblad. Foreign exchange at low, low rates. http://ekblad.cc/ifl15.pdf, 2015.
→ 1 citation on page: 51

[21] A. Ekblad. The Haste language website. http://haste-lang.org, 2015.
→ 1 citation on page: 68

[22] A. Ekblad and K. Claessen. A seamless, client-centric programming model for type safe web applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 79–89, New York, NY, USA, 2014. ACM.
→ 1 citation on page: 77

[23] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.
→ 3 citations on 3 pages: 62, 91, and 94

[24] A. Fritze. The Conductance application server. http://conductance.io, 2014.
→ 1 citation on page: 8

[25] G. Guthrie. Your transpiler to JavaScript toolbox. http://luvv.ie/2014/01/21/your-transpiler-to-javascript-toolbox/, 2014.
→ 1 citation on page: 2

[26] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th*

Annual ACM Symposium on Applied Computing, SAC '14, pages 1663–1671, New York, NY, USA, 2014. ACM.
→ 1 citation on page: 16

[27] D. Herman, L. Wagner, and A. Zakai. The ASM.js draft specification. http://asmjs.org/spec/latest/, 2014.
→ 3 citations on 3 pages: 4, 15, and 17

[28] J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.
→ 1 citation on page: 22

[29] F. Indutny. The bn.js library. https://github.com/indutny/bn.js.
→ 1 citation on page: 29

[30] F. Loitsch and M. Serrano. Hop client-side compilation. *Trends in Functional Programming*, 8:141–158, 2007.
→ 1 citation on page: 42

[31] P. Lubbers and F. Greco. HTML5 web sockets: A quantum leap in scalability for the web. *SOA World Magazine*, 2010.
→ 1 citation on page: 77

[32] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.
→ 1 citation on page: 68

[33] G. Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM.
→ 1 citation on page: 7

[34] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork: an abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 325–337. ACM, 2014.
→ 2 citations on 2 pages: 10 and 17

[35] S. Marlow and S. P. Jones. The new GHC/Hugs runtime system. *URL http://research. microsoft. com/apps/pubs/default. aspx*, 1998.
→ 2 citations on page: 23

[36] S. Marlow and S. P. Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 4–15, New York, NY, USA, 2004. ACM.
→ 3 citations on 3 pages: 25, 26, and 44

[37] S. Marlow, A. R. Yakushev, and S. Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 277–288, New York, NY, USA, 2007. ACM.
→ 1 citation on page: 22

[38] M. Naylor and C. Runciman. The reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In *Implementation and Application of Functional Languages*, pages 129–146. Springer, 2008.
→ 1 citation on page: 2

[39] V. Nazarov, H. Mackenzie, and L. Stegeman. GHCJS Haskell to JavaScript compiler. https://github.com/ghcjs/ghcjs, 2015.
→ 6 citations on 6 pages: 2, 5, 19, 45, 68, and 75

[40] W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.
→ 1 citation on page: 44

[41] T. Petricek and D. Syme. AFAX: Rich client/server web applications in F#. 2007.
→ 2 citations on 2 pages: 2 and 9

[42] S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, 1992.
→ 1 citation on page: 31

[43] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction*, 180:47–96, 2001.
→ 1 citation on page: 92

[44] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, volume 1, pages 203–233, 2001.
→ 1 citation on page: 61

[45] A. Pignotti. Cheerp: a C++ compiler for the web going beyond emscripten and node.js. http://leaningtech.com/cheerp/blog/2013/10/31/Cheerp-Released/, 2013.
→ 1 citation on page: 9

[46] R. Plasmeijer and M. van Eekelen. Clean language report version 2.1, 2002.
→ 1 citation on page: 5

[47] D. Rajchenbach-Teller. Opa: Language support for a sane, safe and secure web. *Proceedings of the OWASP AppSec Research*, 2010, 2010.
→ 1 citation on page: 8

[48] A. Reid. Malloc pointers and stable pointers: Improving Haskell's foreign language interface. In *Glasgow Functional Programming Workshop Draft Proceedings, Ayr, Scotland*. Citeseer, 1994.
→ 2 citations on 2 pages: 64 and 66

[49] M. Scheevel. NORMA: a graph reduction processor. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 212–219. ACM, 1986.
→ 1 citation on page: 2

[50] M. Schinz and M. Odersky. Tail call elimination on the Java virtual machine. *Electronic Notes in Theoretical Computer Science*, 59(1):158–171, 2001.
→ 1 citation on page: 43

[51] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2. 0. In *OOPSLA Companion*, pages 975–985, 2006.
→ 1 citation on page: 10

[52] B. Stroustrup. C++11 - the new ISO C++ standard. http://www.stroustrup.com/C++11FAQ.html, 2013.
→ 1 citation on page: 9

[53] D. Tarditi, P. Lee, and A. Acharya. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(2):161–177, 1992.
→ 1 citation on page: 25

[54] S. Wargolet. Google Web Toolkit. Technical report, Technical report 12. University of Wisconsin-Platterville Department of Computer Science and Software Engineering, 2011.
→ 1 citation on page: 8

[55] A. Wirfs-Brock. ECMAScript 2015 language specification. http://www.ecma-international.org/ecma-262/6.0/, 2015.
→ 1 citation on page: 24

[56] D. Wirtz. The long.js library. https://github.com/dcodeIO/long.js.
→ 1 citation on page: 29

[57] A. Yakeley. The *time* package. http://hackage.haskell.org/package/time, 2014.
→ 1 citation on page: 51