Thesis for the degree of Doctor of Philosophy

Functional EDSLs for Web Applications

Anton Ekblad

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Functional EDSLs for Web Applications
Anton Ekblad

ISBN 978-91-7597-692-1
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4373
ISSN 0346-718X

Technical report 154D
Department of Computer Science and Engineering
Functional Programming Division

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

Printed at Chalmers
Gothenburg, Sweden 2018

Functional EDSLs for Web Applications
Anton Ekblad
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis aims to make the development of complex web applications easier, faster and safer through the application of strongly typed functional programming techniques.

Traditional web applications are commonly written in the de facto standard language of the web, JavaScript, which, being untyped, provides no guarantees regarding the data processed by programs, increasing the burden of testing and defensive programming.

Modern web applications are often highly complex, with multiple interdependent parts interacting over the Internet. Such applications are traditionally implemented with each component as a separate program, exposing its functionality to other components through different API:s over some communication protocol such as HTTP.

This process is mostly manual, and thus error-prone and labour intensive, with accidental API incompatibility between components being particularly problematic. Even in a conventional typed language, the absence of such incompatibilities is not guaranteed. While the different components may well be type-safe in isolation, there is no guarantee that the whole is type-safe as the communication between components is not type-checked.

We present a web application development framework, based on the Haskell programming language, to increase programmer productivity and software quality by addressing these issues. In our framework, an application with an arbitrary number of components is written, compiled and type-checked as a single program, guaranteeing that the application as a whole, including network communication, is type-safe. Communication between components is automatically generated by our framework, eliminating the risk of API incompatibilities completely.

Supporting this framework, we also present a state-of-the-art compiler from Haskell to JavaScript, a novel foreign function interface to allow programs to leverage existing JavaScript code, an embedded language for integrating low-level, high-performance kernels into otherwise high-level web applications, and a highly expressive relational database language.

Keywords: web applications, distributed systems, functional programming, domain-specific programming languages, tierless programming languages

This thesis is based on the work contained in the following papers:

I. A. Ekblad. 2018. Internals of the Haste Compiler. Preprint. https://ekblad.cc/pubs/selda.pdf

II. A. Ekblad. 2015. Foreign Exchange at Low, Low Rates. Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, 2, 2015. ACM.

III. A. Ekblad. and K. Claessen. 2014. A Seamless, Client-Centric Programming Model for Type-Safe Web Applications. In Proceedings of the 2014 ACM SIGPLAN International Symposium on Haskell. ACM.

IV. A. Ekblad. 2017. A Meta-EDSL for Distributed Web Applications. In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. ACM.

V. A. Ekblad. 2016. High-Performance Web Applications through Haskell EDSLs. In Proceedings of the 9th ACM SIGPLAN International Symposium on Haskell. ACM.

VI. A. Ekblad. 2017. Scoping Monadic Relational Database Queries. Preprint. https://ekblad.cc/pubs/selda.pdf

With the exception of Paper III, all work presented in this thesis was conceived, carried out and documented solely by the author.

In the case of Paper III, the problem statement, plus revision work and feedback on the paper itself, was provided by Koen Claessen, whereas the design, implementation and evaluation of the programming model, as well as the writing itself, was carried out by the author.

Contents

Acknowledgements

There are many, many extraordinary individuals who deserve to be credited for their awesome contributions to these past five years of my life and, by extension, to this thesis. It's been a fantastic ride – if slightly bumpy at times – mainly because I've had such a great team to share it with.

First and foremost, I'd like to thank my amazing supervisor Koen Claessen. For your continuous support, for believing in me even when I didn't myself, for encouraging me to chase my ideas, for inspiring me to embark on this journey in the first place, and – last but not least – for all the fun discussions we've had.

To my awesome co-supervisors Emil Axelsson and Alejandro Russo; thank you for your great support and advice!

To Dimitrios Vytiniotis, my supervisor during three intense months at Microsoft Research; for your guidance, for the opportunity to be a part of the Ziria project, and for your invaluable support when I needed it the most – thank you!

To Christine Räisänen and Gerardo Schneider; without your timely advice, this thesis would in all likelihood not exist. To all my colleagues; thank you for making Chalmers such a fun and rewarding place to work. I will miss you all!

To my life partner, Sofia Zaid, and to my other family and friends; your support is the foundation upon which this thesis was built.

Introduction

Functional EDSLs for Web Applications

This thesis explores the application of strongly typed functional programming and embedded language techniques to the domain of distributed web applications. By leveraging the existing ecosystem of the *Haskell* programming language in the creation and popularisation of novel programming models, we hope to improve the safety, ease of development and time to market of modern, distributed web applications.

1 Introduction

In the production of this thesis, no less than three web applications were used for feedback and communication, two for compiling the bibliography, and another three for handling the paperwork. In fact, most of us make extensive use of web applications to manage most aspects of our lives.

It is easy to see the appeal of a web application over a conventional desktop application, both from a user and a developer perspective. For the user, web applications don't require installation, only requiring the user to point their browser to an address and supply their credentials. Connectivity is included by default, and the user's data travels with them from device to device. For the developer, being able to code against a (relatively) homogeneous browser environment is vastly preferable to testing a wide variety of hardware and operating system combinations. Any number of platforms can be supported as long as each has a web browser, and per-user accounts for an online service make it easy to prevent piracy and divide the service into separate versions with different pricing models.

Anatomy of a web application A typical web application consists of three *tiers*: frontend, backend and database. The exact division of tasks between tiers varies considerably depending on the application's problem domain, requirements for latency or offline operation, and other factors. As a rough guideline, however, the frontend mainly interfaces with the user, the backend deals with authentication, communication and computation, and the database deals with data storage and retrieval.

Traditionally, web applications have been built in a monolithic manner: a single client program talking to a single server program over HTTP, using some ad hoc API devised specifically for the application. More recently, a design methodology known as *microservices* has gained popularity: breaking the server monolith up into multiple small services according to the single responsibility principle. The client may then communicate with any number of these small services, and the services may of course also communicate with each other [Namiot and Sneps-Sneppe, 2014].

Common to both methodologies is that the client and the server(s) are treated as independent applications who only happen to be talking to each other. For better or for worse, this property means that web applications are often less cohesive than their more conventional counterparts. That the three tiers of a web application are commonly implemented using different languages makes this divide even wider.

For applications completely or partially implemented using typed programming languages in particular, this has far-reaching implications: while client and servers may well be type-safe in isolation, the application as a whole is not, as its constituent parts are all built and type-checked separately from each other. Problems which can be seen as type mismatches, which in theory could have been caught by a compiler with relative ease, thus instead morph into runtime errors in the communication code.

Tierless web languages To remedy this situation and improve the type-safety of web applications, a multitude of so-called *tierless languages*, such as *Opa* [Rajchenbach-Teller, 2010], *Ur* [Chlipala, 2010], and *Links* [Cooper et al., 2007], were devised. These languages allow developers to implement a web application as a single program, using the same language for all tiers and extending the guarantees provided by the type checker to the whole application. While providing an interesting view of the problem and possible solutions, tierless languages have so far not seen significant use by developers. This is perhaps due to the fact that each language starts out from a "clean slate", with no community or ecosystem of their own.

Another, related, strain of research concerns itself with retargeting existing languages towards tierless programming, either by extending the core language or by using existing language features. Languages such as *Eliom* [Radanne et al., 2016], *Conductance* [Fritze, 2014] and *AFAX* [Petricek and Syme, 2007] take the first approach to OCaml, JavaScript and F# respectively, while this thesis concerns itself with applying the second approach, in a mainly Haskell context.

2 Contributions

This thesis describes a set of novel programming techniques for implementing rich, distributed web applications:

- a JavaScript-targeting compilation scheme with accompanying compiler, which significantly outperforms the current state of the art;

- a thorough analysis of common implementation techniques for web-targeting compilers and their impact on the performance of the aforementioned compilation scheme;
- the *Haste.App* programming model for type-safe, distributed web applications;
- a method for implementing self-optimising, high-performance EDSLs for web applications; and
- a solution to the long-standing problem of correctly scoping queries in a monadic database language.

Each technique is accompanied by a proof-of-concept implementation, which is available from the author's website as free software.

Each contribution is described in detail in one chapter of this thesis, with the exception of the *Haste.App* programming model which is covered in two chapters. Each chapter corresponds to a paper, where papers II through V have been published in the peer-reviewed proceedings of various conferences, and papers I and VI are still undergoing preparation for publication.

The remainder of this chapter gives a brief breakdown of each paper, as well as a statement of contributions for each.

2.1 Foundations for Client-Side Haskell Web Applications

Paper I Paper I seeks to corroborate the hypothesis that lazy, functional languages benefit substantially from relying heavily on complex functionality built into the target language, instead of using a straightforward adaptation of a compilation scheme originally targeting a low-level instruction set, when compiled to another high-level language. Most interpreters and JIT compilers for high-level languages contain extensive optimisations for common code paths. Consequently, a related hypothesis of this paper is that hitting said "hot paths" is an important factor in achieving good performance from a compiler targeting such languages.

To this end, paper I develops a compilation scheme and runtime system for a Haskell dialect targeting the JavaScript language rather than a more traditional machine architecture. In the paper, we present a compilation scheme from the STG [Peyton Jones, 1992] intermediate language of the GHC Haskell compiler to JavaScript. In accordance with the aforementioned assumptions, the compilation scheme performs a high-level translation of the STG language into JavaScript. Matching higher-level data and control structures in STG to higher-level structures in the target language, which are less general than the simple branches and jumps usually produced during code generation, is used to convey higher-level assumptions and invariants directly from the source program to the target language's optimiser.

Unlike when compiling for a native target, and unlike the current state of the art, we do not attempt a lower-level compilation more true to the STG abstract machine. Instead, we rely on the interpreter of the target language being able to recognise high-level code and apply *its own* optimised translation of the target program. A particularly interesting instance of this is explicitly *avoiding* generating tagless code from the STG input, in spite of conventional wisdom, on the assumption that our target language will be able to produce more efficient code from a single branch on a tag than from a more general, considerably heavyweight, function call into a thunk.

Through comparison with GHCJS, the current state of the art in web-targeting compilers for lazy, functional languages, we demonstrate that our compilation scheme produces code which not only runs significantly faster, but is also up to an order of magnitude *smaller*, than code produced by competing compilation schemes. We also perform a survey of relevant optimisation techniques and their impact on the performance of the generated code.

Our work mitigates the problem of lazy, functional programs often being both slow and large. While large code size and significant slowdowns may be acceptable in native binaries, which often have an abundance of processing power and disk space at their disposal, increasing the amount of program code transferred by a large web application by a factor of 10 or even more can be prohibitively expensive. By adopting the compilation scheme proposed in Paper I, implementors of lazy, functional languages can significantly reduce costs to users adopting their languages for use in client-side web applications. The rest of the thesis uses our compilation scheme and the compiler implementing it as a building block for higher-level programming techniques.

Our compilation scheme is implemented in the *Haste* compiler, which is available as free software at `https://haste-lang.org`.

Paper I is a pre-print version of a paper being prepared for submission to the Journal of Functional Programming (JFP), 2018. It is based on the first chapter of the author's licentiate thesis [Ekblad, 2015].

Paper II This paper details the design and implementation of a novel foreign function interface for a functional language compiling down to some higher-order language. Again, the particular languages used as the source and target languages for our reference implementation are Haskell and JavaScript respectively.

In the spirit of the compilation scheme described in Paper I – exploit your host environment as much as possible – this interface uses the target language's own lambda abstractions to represent foreign code. Unlike traditional foreign function interfaces, this has the advantage of allowing arbitrary code fragments, not just named functions, to be imported from the target language into the guest language.

Also unlike traditional foreign function interfaces, our interface allows automatic marshalling of arbitrary data, including higher-order functions, between the host and guest languages. Marshalling of non-function data employs a generic traversal, converting source language data structures to structurally equivalent target language data structures, with the exception of "primitive" types – integers, booleans, etc. – which are losslessly converted between their source and target language primitive representations. Function marshalling is slightly more complicated, dynamically allocating new function objects as needed to convert between source and target language calling conventions.

These two key features taken together allows code written using a foreign function interface based on our technique to be significantly more readable, and to avoid a considerable amount of boilerplate. As an example, consider the following code fragment to fetch the current time, written using the standard Haskell foreign function interface:

```
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 stdcall unsafe "time.h gettimeofday"
   gettimeofday :: Ptr CTimeval → Ptr () → IO CInt

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

Not only is it plagued by considerable amounts of boilerplate code, but much of that code is low-level enough to be nigh incomprehensible without detailed knowledge of the underlying API. The following code fragment performs the same task, but is written using the reference implementation of our interface:

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

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

Contrasting this code fragment with the previous one, we see that not only is the code considerably shorter, but the level of abstraction has also been raised significantly.

In the paper, we describe the design of the interface and give a reference implementation, *fully embedded in the source language*: the target language's dynamic code evaluation facilities are exploited to pass code fragments directly to the its interpreter at run-time, through the source language's built-in, low-level foreign function interface. This property provides significant ease of improvement and experimentation over other approaches, which are normally integrated tightly into the compiler itself. We also demonstrate how the reliance on dynamic code evaluation can be avoided through a minor compiler augmentation. Finally, we give examples to demonstrate that our interface does indeed reduce boilerplate code by a significant amount, and we show through a set of benchmarks, compared against Haskell's standard foreign function interface, that the performance impact of using our interface is in most cases negligible despite its increased expressiveness.

While most languages, Haskell included [Chakravarty, 2003], include perfectly workable interfaces for integrating with other languages, most such interfaces were devised with the intention of interacting with C; the programming lingua franca. Consequently, said interfaces focus on a low-level core API over bytes and pointers. While inconvenient, this detour via a lowest common denominator is often a necessity to bridge the gap between two communicating higher-level languages. However, when compiling to a higher-level language, said higher-level language may instead be used as the lowest common denominator. At this point, the low-level detour is not only inconvenient, but even *increases* the compatibility gap between higher-level languages. Our foreign function interface elegantly solves this problem while remaining reasonably performant.

This foreign function interface serves as the cornerstone for the reference implementations of the findings presented in papers III through V. Consequently, all implementations can be used with any JavaScript-targeting Haskell compiler, after implementing this interface. As the interface's only compiler-specific components are the stubs for the compiler's native foreign function interface and a small piece of target-language JavaScript for marshalling functions using the source language's particular calling convention, porting the interface to any such compiler is straightforward.

Paper II is based on a paper presented at the Symposium on the Implementation and Application of Functional Languages (IFL), 2015.

Impact The Haste compiler and its accompanying foreign function interface have been used in several functional programming courses at Chalmers University of Technology and one MSc. thesis [Sjösten, 2015]. A BSc. thesis at Chalmers [Block et al., 2016] carried out a more systematic evaluation of the Haste compiler and the Haste.App programming model described in Paper III, in regards to usability and performance, with a generally favourable outcome.

The Haste compiler has seen some use in industrial settings, and the ideas underpinning the FFI described in Paper II have made their way into industry on their own; for instance, the foreign function interface of the Haskell embedding of the *R* language by Tweag I/O [Boespflug, 2015] is explicitly based on Paper II.

The Haste compiler has also garnered some attention in open source circles, so far totalling more than 16 000 downloads from the Hackage package repository alone, with another conservatively estimated 10 000 binary and source code downloads from the project's website and GitHub source repository. Said repository is at the time of writing the 25th most "starred" [1] out of the more than 60 000 Haskell projects on GitHub. Talks about the compiler have been given by independent third parties at industry conferences such as BayHac, CampJS and Strange Loop, in addition to an invited talk by the author at the MLOC.JS web development conference [Ekblad, 2014, Kuhtz, 2014, Miller, 2014, Swenson-Healey and Cooper, 2014].

2.2 Type-Safe, Distributed Web Applications

Paper III This paper presents a tierless programming model for implementing rich, client-server web applications using only standard Haskell,

[1] A measure of popularity, akin to a Facebook "like".

and the *Haste.App* library implementing it. Web applications, traditionally implemented as two or more independent programs communicating using some ad hoc communication protocol, are in this model written as a single Haskell program. The client and server parts of the application are separated by the type system, with client-side code residing in a *Client* monad and server-side code in a corresponding *Server* monad. The computation is driven by the client side, with the server lying dormant until a client makes a remote procedure call to some function in the Server monad; the server may not call back into the the client on its own volition. This restriction helps keep the program flow clear and explicit.

A key component of this programming model is that programs are compiled not once, but *twice* – once to produce a server binary, and once to produce client-side JavaScript code. The supporting *Haste.App* library splits the application in two parts, ensuring that code executing in the Client monad ends up on the client, and that code executing in the Server monad ends up on the server. A third monad is used as a staging area, where server-side functions are "imported" onto the client, to provide the glue between the client and the server. This code is executed on the client as well as the server, but performs a slightly different task depending on where it executes: on the client, it connects source-language identifiers to their corresponding server-side RPC endpoints, whereas on the server, it builds a lookup table to map client RPC calls to their server-side implementations. Pure code – that is, effect-free code callable from any monad – is duplicated, and ends up on both client and server.

This automatic splitting is achieved by compile-time introspection. When the library detects that the program is being compiled for the client, it filters out any server-side code, replacing all calls to such functions with stubs taking care of the network communication and synchronisation necessary to make a remote call. Similarly, when the library detects that the program is being compiled for the server, all client-side code is filtered out and the program's entry point is replaced by an event loop, dispatching server-side functions to fulfil remote calls as they come in from clients.

As previously mentioned, there exists a wealth of previous work on tierless web applications. What sets our programming model apart is its reliance only on existing tools for the Haskell language: it can be implemented entirely as a library, without the need for new languages or compiler modifications. In addition to lending itself well to agile experimentation, this has the benefit of letting it ride on the coattails of the Haskell community and infrastructure, as developers can combine it with their favourite Haskell tools, libraries and idioms right out of the box. A key insight in enabling this is the use of multiple compilers to produce different binaries from a single program, with the supporting Haste.App library controlling the placement of code fragments.

Paper III is based on a paper presented at the Haskell Symposium, 2014, coauthored with Koen Claessen.

Paper IV While the programming model presented in paper III works well for single-server applications, its model of a web application is overly simplistic. Commonly, an application does not consist of a client and a single server, but of a client and *multiple* servers, which are often heterogeneous in nature. Computational resources, databases, ad servers, etc. are all common occurrences in present-day web applications.

In paper IV, we generalise the results from paper III to cover an arbitrary number of interconnected servers. We keep the basic premise of using two compilers to produce client and server executables, but generalise it to allow any number of different binaries to be produced by any number of compatible compilers. We also keep the idea of determining code placement based on the type of an expression, but again generalise this concept from two designated client and server monads to any number of different nodes. Communication between nodes is still handled like "normal", type-safe, monadic function calls, with the underlying network code being generated by the supporting library.

We also lift the requirement that nodes must be monadic, allowing, for instance, applicative and arrow nodes. This is partially to better support *exotically typed* nodes – nodes on which calculations are performed in another type universe than on the calling client. Tight integration with exotically typed nodes reduces the amount of boilerplate code required to directly connect nodes written in some embedded, domain-specific language to the network, as even EDSLs with very different programming models – SQL queries or GPU kernels, for instance – can be called by clients without having to deal with type conversions or the details of how to compile, load and execute the embedded programs.

Nodes are connected in a directed graph, where the client node – the one executing in the user's browser – is transitively connected to all other nodes. The client still drives the computation, but server nodes may now themselves be clients of *other* server nodes as well. A node is connected to the network by instantiating a type class, describing how it may be reached by other nodes, which other nodes may communicate with it, how its type universe maps to that of any calling client, and whether the node needs any particular initialisation. For nodes which are not exotically typed, most of these properties are optional, defaulting to values appropriate for most nodes written in "normal" Haskell.

We further generalise the concept of nodes to cover virtual servers as well, and demonstrate how this lets us model fine-grained sandboxing of untrusted JavaScript code as simple RPC calls. Web applications often load third party code from external sources at runtime, which makes them vulnerable not only to being compromised themselves, but to security breaches on any network or remote host from which code is loaded. Our sandboxes-as-servers method exploits browsers' built-in sandboxing mechanisms, which are normally too coarse-grained and cumbersome to see widespread use, to implement convenient, fine-grained sandboxing.

These generalisations sets Haste.App further apart from other tierless web programming models, which generally only support a fixed set of nodes under a fixed set of programming paradigms.

Paper IV is based on a paper presented at the Haskell Symposium, 2017.

Impact Like the Haste compiler, a systematic evaluation of the initial Haste.App programming model, described in paper III, was carried out at Chalmers University of Technology, with favourable results both regarding performance and usability. While industry interests have been rather more reluctant to embrace Haste.App than the Haste compiler, it has been used with positive results to implement research and teaching software at Chalmers as well as at other institutions [Kahl, 2016].

2.3 Domain-Specific Problem Solving through EDSLs

Paper V This paper describes a method for integrating high-performance, low-level computational kernels into Haskell web applications, and an accompanying proof of concept EDSL.

While concise, high-level and readable code is often preferable to highly performant code for most applications – developer time being significantly more expensive than processing time – some applications may benefit from having both. For instance, online games, signal processing and cryptographic applications may all have performance-critical bottlenecks where the performance penalty imposed by higher-level languages is unacceptable, while large parts of the application – such as user interfaces and data storage – may not be very performance sensitive at all. In a traditional setting, such situations are often resolved by writing the performance-critical parts of an application from C and integrating them into the higher-level application using some foreign function interface, but in a web application there is no such recourse.

Paper V provides a solution to this problem, in the form of the *Aplite* low-level EDSL geared towards computationally heavy tasks. We build on the results from the DSP-targeting Feldspar language [Axelsson et al., 2010], but adapt the methods to a web context, exploring a multi-backend compilation scheme targeting both *ASM.js* – a subset of JavaScript designed to be highly optimisable – and plain but efficient JavaScript.

We demonstrate a method to seamlessly integrate Aplite kernels into the Haskell host program, making them indistinguishable from "normal" Haskell functions, even though Aplite kernels have a completely different type universe from its Haskell host. Kernels with host-observable side-effects can be imported as plain Haskell functions in the IO monad, whereas kernels with only local side-effects may be imported either as pure or monadic functions, depending on which type best suits the programmer. This is accomplished by leveraging the dynamic code loading capabilities of the foreign function interface described in paper II and judicious application of type-level functions. Being first-class objects, Aplite programs represent an application of multi-stage programming, allowing the host program to specialise Aplite programs to its runtime environment as well as user input and other parameters.

Recognising that different applications may have wildly different performance characteristics depending on the combination of backend, browser environment and user input they are presented with, Aplite supports re-compiling existing kernels with different parameters. More interestingly, we demonstrate a method whereby a kernel is *automatically* profiled with a series of different compilation parameters, and the most efficient implementation selected by any subsequent calls to the kernel.

We thoroughly benchmark our reference implementation against web-targeting Haskell code as well as hand-rolled JavaScript, and demonstrate that our language outperforms both on all investigated benchmarks. In particular, we demonstrate that backend selection must be informed by both the performance characteristics of the kernel in question and the current browser environment, giving solid evidence for the efficacy of our multi-backend compilation scheme and profile-guided backend selection.

Paper V is based on a paper presented at the Haskell Symposium, 2016.

Paper VI This paper presents a simple but effective solution to the long-standing problem of ensuring the well-scopedness of a monadic formulation of relational database queries, together with a simplified version of *Selda*, the first relational database EDSL to support both a well-scoped monadic interface and fully general[2] inner queries.

While there exists an ample body of previous work in EDSLs for integrating with relational databases, so far none has managed to ensure that queries are well-scoped in the presence of fully general inner queries, while

[2]As opposed to static SELECT statements over fixed tables.

maintaining a monadic interface. Monadic interfaces are useful for Haskell EDSLs, as they provide a familiar and well understood interface to user and developer alike, with good support from standard and third-party libraries.

Consider the following monadic pseudocode query, intended to associate each person with their home city, but only if said city is located in Sweden.

```
addresses = do
  (name :*: addr) ← select persons
  city ← leftJoin (\city → addr .== city) $ do
    (city :*: country) ← select cities
    restrict (country .== "Sweden")
    return city
  return (name :*: city)
```

A straightforward translation into SQL presents us with the following query.

```
SELECT personName, cityName
  FROM persons
  LEFT JOIN (
    SELECT cityName
    FROM cities
    WHERE cities.country = "Sweden"
  )
  ON persons.address = cityName
```

While this query is not problematic per se, the fact that the *name* and *addr* identifiers are in scope *inside the body of the left join* is a cause for concern. In fact, this enables the creation of decidedly nonsensical queries, as in the following modification of the above example.

```
illScopedAddresses = do
  (name :*: addr) ← select persons
  city ← leftJoin (\city → addr .== city) $ do
    (city :*: country) ← select cities
    restrict (country .== "Sweden")
    restrict (city .== addr)
    return city
  return (name :*: city)
```

Here, the body of the join refers directly to the *addr* identifier, even though no table referenced by the inner query has any such field; the query is *ill-scoped*. Clearly, any type-safe relational database EDSL must disallow such nonsensical queries.

This paper presents a simple way to ensure the well-scopedness of inner queries based on type-level functions over phantom types. Like the standard Haskell *ST* monad [Launchbury and Peyton Jones, 1994], Selda solves the problem by parameterising its query monad over a phantom type denoting its scope. Expressions in the monad are then also augmented with a scope parameter, ensuring that computations can only operate on expressions within its own scope.

Unlike the ST monad, which only allows pure values to be returned from stateful computations, inner queries in a database EDSL must be able to return *EDSL expressions* to the outside world. As this is not possible using the method employed by the ST monad, its type parameter being existentially quantified, we instead view the scope parameter as a *scope counter*. The outermost query has a scope equivalent to zero, and each nesting of an inner query increments the scope counter by one. Expressions returned from an inner query have their scope counter decremented by one, to allow the outer query to operate on them, but crucially, expressions are *not* able to migrate *inward*, solving the scoping problem introduced previously.

We also discuss the similarities and differences between general inner queries and inner aggregate queries, show how a similar problem arises when compiling aggregated queries to SQL, and demonstrate how the scope counter solution may be applied to the aggregate compilation problem. Finally, we present a simplified version of the Selda API, demonstrating how the scope counter solution can be incorporated in the language to provide a simple, monadic interface that supports fully general inner queries while ensuring their well-scopedness.

Paper VI is based on a paper presented at the Symposium on Trends in Functional Programming, 2017, and being prepared for submission to the 2018 Haskell Symposium.

Impact While not as popular as the Haste compiler, the Selda library has in its nine months of existence become the fourth most downloaded database EDSL on the Hackage package repository, as well as one of the 200 most popular Haskell repositories on GitHub out of more than 60 000. As paper VI is as yet unpublished academic interest has been scarce, but the community surrounding the library includes several industrial users.

Aplite, on the other hand, has so far not garnered any significant industrial or open source interest. With the rapid advance of WebAssembly [Eich, 2015] rendering ASM.js largely obsolete, this is not expected to change without significant retargeting and repackaging efforts.

3 Background

3.1 Functional Programming

Functional programming is a discipline of software development which views programs as functions from inputs to outputs, built from smaller functions which are in turn built from *even smaller* functions, and so on. Unlike the more familiar functions encountered in high school, a functional program does not restrict itself to operations over, say, real numbers. Mouse movements, real-time audio streams and even other programs are all examples of possible inputs, while outputs may include the sending of messages over a network, pixels displayed on a screen, or haptic feedback through a game controller.

Functional programs are declarative: the programmer describes the relations between the application's states, focusing on *what* the application is supposed to be doing. In contrast, imperative programs consist mainly of code describing *how* the program is intended to achieve its goal.

Higher-order functions The *functional* in functional programming is perhaps most apparent in its treatment of functions as *first-class objects*: just like integers or floating point numbers, functions are just another type of data to be created, passed around and bound to identifiers – or not, as the programmer chooses. This enables powerful forms of decoupling and abstraction, where functions may leave "holes" of undefined behaviour, to be filled in by its caller.

For instance, consider the following implementation of the `merge` function, which merges two lists which are sorted in ascending order, into a single list which is also sorted in ascending order:

```
merge :: Ord a ⇒ [a] → [a] → [a]
merge (x:xs) (y:ys) =
  if x < y then x:merge xs (y:ys)
           else y:merge (x:xs) ys
merge _ [] ys = ys
merge _ xs [] = xs
```

While this function is certainly useful, perhaps to display two separate, ordered data sources to a user as a single table, it is not very flexible: what if we sometimes need to merge two lists sorted in *descending* order? We could, of course, implement *two* functions – `mergeAscending` and `mergeDescending` – but by the *DRY*[3] principle [Hunt and Thomas, 2000], we really shouldn't.

A better solution would be to parameterise the function's behaviour over the way in which we want to sort the elements:

[3] Don't Repeat Yourself

```
merge2 :: Ord a ⇒ Bool → [a] → [a] → [a]
merge2 ascending (x:xs) (y:ys) =
  if ascending
    then if x < y then x:merge2 ascending xs (y:ys)
                  else y:merge2 ascending (x:xs) ys
    else if x > y then x:merge2 ascending xs (y:ys)
                  else y:merge2 ascending (x:xs) ys
...
```

While this solution certainly contains less repetition than writing two separate functions, it is still not ideal. We don't entirely get rid of repetition and, most importantly, *we can only support merging behaviours that the original implementer of `merge2` could foresee!* This is a real problem when working with data that does not necessarily have a single, canonical total ordering but which we still may want to sort somehow: tuples of numbers sorted by some mathematical property, or cartoon ponies sorted by their suitability for some given task, for instance.

Instead, in a functional program we would parameterise the merge function, not over a flag to choose one of several hard-coded comparison behaviours, but *over a function describing the comparison itself*:

```
mergeBy :: Ord a ⇒ (a → a → Bool) → [a] → [a] → [a]
mergeBy goesBefore (x:xs) (y:ys) =
  if x ‘goesBefore‘ y then x:mergeBy goesBefore xs (y:ys)
                      else y:mergeBy goesBefore (x:xs) ys
...
```

By simply leaving the choice of the comparison function up to the caller, we gain several important advantages: we no longer need to implement different behaviours depending on some user-supplied flag, we get rid of the repetition inherent in doing so, and – most importantly – we separate the task of merging two lists from the task of comparing two elements.

Functions that accept other functions as inputs are known as *higher-order functions*, and are the bread and butter of functional programming, providing programmers with a natural means to modularise their programs, with any desired level of granularity.

Function composition The treatment of functions as first-class objects enables us to write higher-order functions to *compose* functions in various, often highly generic, ways. As a example, Haskell provides a standard function composition operator to compose two functions `f` and `g` by creating a new function which first applies `g` to its argument `x`, and then applies `f` to the result of `g`:

```
(.) :: (b → c) → (a → b) → (a → c)
f . g = \x → f (g x)
```

Almost trivial in its definition, standard function composition is surprisingly useful, allowing many complex functions to be expressed as a *pipeline* of smaller, less complex, functions.

As an example, consider the following function:

```
toSet :: (Ord a, Eq a) ⇒ [a] → [a]
toSet = map head . group . sort
```

This function performs the nontrivial task of turning an unsorted list of possibly duplicate elements, into an ordered set: a list which is sorted and guaranteed to contain no duplicate elements. Instead of tackling the whole task at once, we construct the solution as a pipeline: first we `sort` the input list, then we `group` all adjacent elements that are equal to each other into sub-lists, and finally we extract the `head` – or first element – of each such sublist.

Compared to a more monolithic solution, it is easy to convince oneself that this function is correct: if two or more elements are equal, then they must all be adjacent to one another after sorting; if two or more equal elements are adjacent to each other, they must all end up in the same sublist after grouping; ergo, extracting the first element of each such sublist trivially gives us a single representative for each group of equal elements.

Note how this manner of programming plays into the aforementioned theme of modularity: the `toSet` function is a simple composition of prebuilt, generic functions, with no conditionals or logic of its own save for the choice of functions used in its implementation and the order in which they are composed. This is a great boon to modularity and code reusability, which in turn brings substantial benefits for programmer productivity [Hughes, 1989].

Higher-order functions as OOP design patterns Readers familiar with object-oriented design patterns [Gamma et al., 1993] may notice certain similarities between higher-order functions, function composition, and several design patterns: the *command*, *visitor*, *observer*, *strategy* and *dependency injection* patterns directly correspond to different specialised uses of higher-order functions, while patterns such as *bridge*, *facade*, *adapter*, *builder* and *proxy* can be easily implemented using function composition.

Pure functional programming While functional programming in general can be beneficial to programmers, the Haskell programming language used throughout this thesis takes the functional programming paradigm one step further, in its adoption of *pure functional programming*.

In a purely functional programming language, all functions are functions in the *mathematical* sense: the output of a function depends solely on its inputs. That is to say, a function may not give a different result depending on the number of times it has been called, the current date, or any other information that may vary depending on the program's circumstances. A pure function must also not perform any *effects* – changing the state of the program or its environment – aside from computing its value. Consequently, data in a purely functional program is *immutable*.

Immutability brings a host of benefits for programmers: it is easier to reason about programs without ever-changing program state, and bugs are easier to prevent, diagnose and rectify in a program with fewer "moving parts". The advantages of immutability are widely recognised, and its application is recommended even for languages with relatively poor built-in support for enforcing immutability [Bloch, 2008].

Above all, pure computations are highly composable, as they do not make any assumptions about the context from which they are called, only relying on their inputs. This makes the dependencies of program units explicit, reducing the cognitive burden on the programmer when assessing whether a particular modification will affect another part of the program. This property is particularly important for web applications, which are mainly event-driven and continuation based, making their flow of execution impossible to predict.

Enforcing purity through types While immutability by convention is indeed an important step up from a programming style based on gratuitous mutation, it still leaves many things to be desired. It is not always trivial to convince oneself that a piece of code written in, say, C# or JavaScript is indeed pure: as purity is not tracked across compilation units – or even classes or methods – mutation may hide in the murkiest code depths, many layers of indirection removed from the call sites where we would like to verify its immutability. The allure of sacrificing compositionality by turning to mutation as a quick and dirty fix, with only programmer discipline to keep it in check, only serves to exacerbate this problem.

To fully reap the benefits of purity, Haskell encodes the effects a function may perform in its type, encapsulating computations with side effects in a type of their own called `IO`. If an integer has the type `Int`, then a computation that produces an integer has the type `IO Int`; and if a function which accepts a string as its input and returns an integer has the type `String -> Int`, then a function from strings to integers which *also* may perform some effect has the type `String -> IO Int`. This ensures that purity is tracked and enforced throughout programs: if we have a function `plusOne :: Int -> Int`, applying it to a computation of type `IO Int` rather than a plain `Int` will cause a type error during compilation.

At first glance, this would seem to preclude the use of pure functions to manipulate user input, as it is plainly impossible for a function which reads user input – thereby depending on the state of the world – to be pure. Fortunately, the `IO` type is an example of a *monad* [Wadler, 1995]: an abstraction which allows values contained in another type to be manipulated by pure functions, the resulting new value safely re-encapsulated within the containing type again. Thus, in a Haskell program, impure code may depend on pure functions, but pure code may *not* depend on impure functions.

3.2 Embedded Domain-Specific Languages

Embedded, domain-specific languages, or *EDSLs*, are a software design pattern in which different problem domains of an application are addressed using different programming styles, effectively creating a set of task-specific sub-languages *inside* the general-purpose host language. It is related to the object-oriented *interpreter*, or *little languages* design pattern, where an application outsources some problem domain to an *external* domain-specific language, which the main application then interprets [Bentley, 1986].

The main difference between the two design patterns lies in the *embedded* part: where an external domain-specific language has the freedom to implement any conceivable language, an EDSL instead piggy-backs on the capabilities of the host language. While an EDSL does not enjoy the same degree of freedom as its non-embedded kin, it also does not incur the same implementation and usage overheads. At the cost of being bound by the host language's restrictions, the EDSL gains the use of the host language's parser, type system, runtime system, tooling, and so on.

EDSLs have a strong tradition in the functional programming community [Hudak, 1996], but has also seen significant adoption in other, more mainstream, communities, with high-profile projects such as Tensorflow [Abadi et al., 2016] and RSpec [Chelimsky et al., 2010] being built largely on this idiom, in Python and Ruby respectively.

It can sometimes be hard to make the distinction between an EDSL and a particularly focused library. Ultimately, this is often a matter of branding, and the level of cohesion between the components making up the EDSL/library. In this thesis, we take the view that an EDSL is a library where the components are intended to be used almost exclusively together in a cohesive manner to solve problems in some particular domain, as opposed to being thinly sprinkled across a code base largely consisting of "other" host language code.

EDSLs and types While the EDSL design pattern can be powerful in any language, the dynamic type system and reliance on convention offered by common implementation languages such as Ruby and Python can sometimes, perhaps counter-intuitively, *restrict* the applicability of EDSLs.

As a motivating example, let us look at the Haskell *STM* EDSL, which allows the programmer to implement communication in a concurrent program as a set of *transactions* over shared mutable variables. STM works on the principle that, just like with relational database transactions, most concurrent data accesses do not interfere with each other, and the overhead of locking shared data is thus often unnecessary. Additionally, handling multiple locks at once is a subtle and error-prone business, and is best avoided whenever possible.

Instead, STM programs are separated into transactions, where each transaction reads and writes shared references with impunity, only committing the result of *all* writes at the end of the transaction. If any shared reference accessed by the transaction was modified at some point during the transaction, the result is not committed but the whole computation is instead *retried* until it succeeds.

It is easy to see that this programming model places some heavy restrictions on the programmer: any code placed within a transaction must be free from effects, as the transaction may be retried any number of times before finally being committed. If any piece of the transaction causes, say, an intercontinental missile to be launched, high contention over shared resources may cause us to retry the transaction a significant number of times, thereby exhausting our stockpile of missiles even though we only intended to fire one!

It is equally easy to see that this EDSL would be depressingly unsafe if implemented in a language where purity is not enforced by the compiler. As pointed out in Sect. 3.1, manually ensuring that any piece of code is indeed pure is a non-trivial task – one that we may want to avoid altogether if the correctness of our application depends on it.

In Haskell, by contrast, this property can be almost trivially guaranteed by leveraging the type system. The monad concept used to model effectful computations in Haskell, turns out to be a flexible, general abstraction for implementing EDSLs, providing a greater level of cohesion and isolating EDSLs from each other as well as from impure host language code [Hudak, 1996]. The STM EDSL uses this to great effect by giving all transactions the type `STM a`, where `a` can be any type. By not including an operation in the language to turn an IO computation into an STM computation, the risk of effectful computations being executed more than once due to transaction retries is completely eliminated.

The power of Haskell's type system allows us to restrict the behaviours of embedded programs even further, for instance by disallowing pure computations over types not well suited to the problem domain [Axelsson et al., 2010, Bracker and Gill, 2014, Svenningsson and Svensson, 2013] or by enforcing custom scoping rules [Launchbury and Peyton Jones, 1994].

In essence, when it comes to EDSLs, it is sometimes the case that "less is more", and that by restricting embedded languages to their target problem domain – and that domain *only* – we open up additional opportunities for safe and efficient problem solving.

4 Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

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.

J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

J. Bloch. *Effective Java*. Addison-Wesley, 2008.

B. Block, J. Gustafsson, M. Milakovic, M. Nilsen, and A. Samuelsson. Evaluating Haste.App: Haskell in a web setting. Effects of using a seamless, linear, client-centric programming model, 2016.

M. Boespflug. Haskellr. https://tweag.github.io/HaskellR/, 2015.

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. doi: 10.1007/978-3-319-04132-2_5.

M. M. Chakravarty. *The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*. 2003.

D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy. The RSpec Book: Behaviour Driven Development with RSpec. *Cucumber, and Friends, Pragmatic Bookshelf*, 2010.

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. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806612.

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. ISBN 978-3-540-74791-8. doi: 10.1007/978-3-540-74792-5_12.

B. Eich. From ASM.js to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/, 2015.

A. Ekblad. Hastily paving the way for diversity. http://www.ustream.tv/recorded/43804744, 2014.

A. Ekblad. A distributed haskell for the modern web. 2015. Also available from https://ekblad.cc/pubs/haste-licentiate.pdf.

A. Fritze. The Conductance application server. http://conductance.io, 2014.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

J. Hughes. Why functional programming matters. *The computer journal*, 32 (2):98–107, 1989.

A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

W. Kahl. CalcCheck: A Proof-Checker for Gries and Schneider's "Logical Approach to Discrete Math". http://calccheck.mcmaster.ca/, 2016.

L. Kuhtz. Haste: Front end web development with haskell. https://www.youtube.com/watch?v=Arot_cDmQHI#t=220, 2014.

J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178246.

K. Miller. Make haste: Fast track fo functional thinking. https://www.youtube.com/watch?v=o3JMxnnTZ64, 2014.

D. Namiot and M. Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

T. Petricek and D. Syme. AFAX: Rich client/server web applications in F#. 2007.

S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2: 127–202, 1992. ISSN 1469-7653. doi: 10.1017/S0956796800000319.

G. Radanne, J. Vouillon, and V. Balat. Eliom: A core ML language for tierless web programming. In *Asian Symposium on Programming Languages and Systems*, pages 377–397. Springer, 2016.

D. Rajchenbach-Teller. Opa: Language support for a sane, safe and secure web. *Proceedings of the OWASP AppSec Research*, 2010.

A. Sjösten. SWAP-IFC: Secure Web Applications with Information Flow Control. 2015.

J. D. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 299–304, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500611.

E. Swenson-Healey and J. Cooper. Haste: Full-stack haskell for non-phd candidates. https://www.youtube.com/watch?v=3v03NFcyvzc, 2014.

P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

Paper I

Internals of the Haste Compiler

Abstract

This paper describes the design and implementation of *Haste*, a state-of-the-art, web-targeting Haskell compiler. In contrast to competing attacks on the problem, Haste uses a high-level translation scheme and lightweight runtime, attempting to make maximal use of the infrastructure provided by the web browser.

We give a detailed description of Haste's translation scheme and design philosophy, discuss several optimisations and their impact on web-targeting code in particular, and demonstrate that Haste produces code which is significantly faster *and* smaller compared to the current state of the art, by factors of up to 2.4 and 10 respectively.

1 Introduction

During the last decade, web applications have taken the software world by storm, to a degree where the very idea of making it through a whole day without interacting with one now almost seems ludicrous.

Interactive web applications are commonly implemented using JavaScript, the only language universally supported across web browsers. However, JavaScript is considered by many to be quirky, unsafe, and even unsuitable for large scale application development [Ekblad, 2012]. In particular, JavaScript's lack of type safety can be considered a significant problem, particularly in functional programming circles.

Instead, we would like to write our web applications using expressive type systems and functional patterns, preferably using an already established language. While there already exist several more or less functional languages targeting JavaScript specifically – such as Fay [Done, 2012] and PureScript [Freeman, 2016] – there is a lot of uncovered ground in the design space for web-targeting compilers for functional languages.

This paper describes the design and implementation of the *Haste* Haskell-to-JavaScript compiler, comparing it to the current state of the art, and exploring the merits of its translation scheme and design choices compared to other approaches.

1.1 Background

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 specialised computer architectures such as the Normal Order Reduction MAchine (NORMA) [Scheevel, 1986]. 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 [Naylor and Runciman, 2008].

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 [Benton et al., 1998]. With the rise of the web as an application platform, just about every functional language now has a compiler targeting JavaScript: LuvvieScript [Guthrie, 2014] for Erlang, Ocsigen [Balat et al., 2012] for O'Caml, AFAX [Petricek and Syme, 2007] for F#, several for Haskell [Dijkstra et al., 2013, Ekblad, 2012, Nazarov et al., 2015], and so on.

Another Haskell-to-JavaScript 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 optimising compiler from Haskell to JavaScript, complete with custom garbage collection, JavaScript optimisation and runtime facilities, would demand quite significant effort – well beyond the scope of this paper. Instead, the Haste compiler adopts, somewhat tongue in cheek, a single guiding principle: *it's someone else's problem*.

Considering this principle, it may seem odd to implement a Haskell to JavaScript compiler at all. After all, both the GHCJS [Nazarov et al., 2015] and UHC [Dijkstra et al., 2013] 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, size matters greatly 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 compiled with GHC [Ekblad, 2012], making it a poor starting point for a JavaScript backend, which will invariably have a performance handicap relative to any native compiler even under the best of circumstances. 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 a more promising starting point. Based on GHC, it boasts excellent compatibility with language features and third party code, and produces relatively fast programs. However, GHCJS also produces relatively large output. 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.

Contributions Our work makes the following contributions to the field: In sections 2 through 5, we describe an efficient compilation scheme and data representation for compiling lazy, functional languages to JavaScript. In section 6 we give a detailed analysis of several optimisations specifically relevant to machine-generated JavaScript code and how they relate to our compilation scheme. Finally, in section 7 we compare our translation scheme to the current state of the art across several benchmarks, demonstrating that our scheme reduces code size by up to a factor of 10, and reduces execution time by up to a factor of 2.4.

2 Overview of the Haste Compiler

Haste explores a different point in the design space: instead of emulating vanilla GHC to the extent possible, it aims to strike a balance between staying faithful to GHC and producing code which is palatable to most common JavaScript engines. JavaScript engines are complex beasts, heavily optimised for code commonly encountered in real applications. Haste's design hypothesis is that a translation scheme approximating this kind of "normal-looking" code can gain a significant performance advantage over translation schemes mainly designed for compilation to native code.

2.1 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.

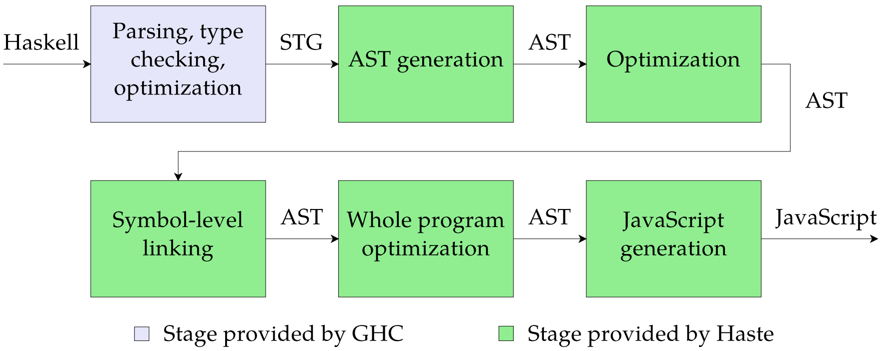

Figure 1: The Haste compilation pipeline.

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 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 optimisations 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 optimisations.

- The optimised 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 optimisations are performed at this stage.

- Finally, the optimised 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, JavaScript has several idiosyncrasies when compared to traditional low level compilation targets: the lack of pointers, integer arithmetic and unstructured jumps, and the presence of high-level features such as first class functions and native garbage collection. 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, 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 optimisations implemented by GHC have been applied at this stage and thunk evaluation has been made explicit.

3 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.

3.1 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 [Hughes, 1989]. 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 optimisations used by vanilla GHC [Marlow et al., 2007].

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 [Marlow and Peyton Jones, 1998]. 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 exists 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 behaviour 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 optimises 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 [Marlow and Peyton Jones, 1998].

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 $t.x$.

3.2 Proper tail recursion

In functional programs, *proper tail recursion*, or *tail call optimisation* 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 an unbounded number of 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 function 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 optimisation.

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 optimisation. The quite recently ratified sixth edition of the *ECMA-262* standard introduces native support for tail call optimisation 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 [Wirfs-Brock, 2015]. Additionally, browser vendors have so far proved reluctant to implement tail call elimination, citing concerns over performance and debugging workflows [Klein, 2017].

Until this situation is sorted out, 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 [Tarditi et al., 1992]. 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 optimisations, it is possible to reduce the performance impact of trampolining. The measures taken by Haste to this end are described in section 6. The significant performance boost obtained when optimising and, where possible, removing the trampolining instrumentation lends some credibility to the above assumption that trampolining takes a relatively heavy toll on performance.

3.3 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 $f : a \rightarrow b \rightarrow c \rightarrow d$ to one, two or three arguments; the only difference is in the return type being $b \rightarrow c \rightarrow d$, $c \rightarrow d$ 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 [Marlow and Peyton Jones, 2004]. 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 conundrum: when all functions are conceptually unary, what does the term "n-ary function" even mean?

For performance reasons, given a function $f : a \rightarrow b \rightarrow c$ 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 $b \rightarrow c$. GHC's optimiser already does a good job determining where this is possible, so an STG lambda expression $\lambda\ x_1\ ...\ x_n \rightarrow ...$ 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 [2004]. 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 less than or equal to the arity of the underlying function. 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.

Generalised 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*:

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

- 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 specialised versions for functions of several common arities, making the actual use of the general application relatively rare.

4 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

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

Table 2.2: Haste type representations

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.

4.1 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 4.2, 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 maths 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 recognises 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* [Wirtz] and *bn.js* [Indutny] libraries to represent 64 bit integers and arbitrary precision integers respectively.

4.2 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

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

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 no 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* 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 3.1, 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 optimisations. It also makes the syntax of their creation slightly wordier, and requires some boilerplate code for each type present in a program.

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

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 7.4.

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

5 From STG to JavaScript

As mentioned in section 2, 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.

5.1 The STG language

STG is more than an intermediate representation: it is a complete abstract machine, with well defined data representations and compilation strategies [Peyton Jones, 1992]. 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. 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:

- 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 optimisation 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 labelled 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 5.3.

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 paper would be rather impractical. The nature of their being basic building blocks also makes the vast majority of

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

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.

5.2 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 2.5 and 2.6 show the statements and expressions of the AST respectively.

Statements A program is, at the top level, a series of assignments terminated by *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*

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

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 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 optimiser 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 3.2 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 3.3, but do *not* handle calls to the trampolined functions described in section 3.2. 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 3.1. 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.

5.3 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 bindings, ret_t for thunks, and ret_f for functions. Post-generation, an optimisation 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 optimisations performed by Haste are described further in section 6.

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 4.
- $V(n)$ 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 3.1.
- $prim(op)$ gives the AST implementation of a primitive operation op. Primitive operations are further discussed in section 5.1.
- $fresh(v)$ 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 (\bnd → S(bnd, Stop))
```

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

Table 2.7: Expression translation

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

Table 2.8: Statement translation

This results in a list of AST bindings, which are then optimised 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` function of a Haskell program, constitutes the input of the linking process described in section 5.4.

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 optimisation passes, and some are used extensively by Haste's implementation of various primitive operations but

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

Table 2.9: *case* expression translation

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-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 optimisation 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 3.2 and 6, 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.

5.4 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 optimisation pass is optionally run over the resulting program, to perform the same inlining and other optimisations as were performed on a per function basis, but this time over the whole program.

6 Optimising JavaScript

Optimisation is one area where Haste's guiding philosophy – it's someone else's problem – shines the brightest. As explained in section 2.1, using GHC's intermediate STG representation as our source language gives us a full suite of state of the art optimisations 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 optimiser – at our disposal. These optimisations apply to Haskell and GHC's intermediate formats in general; there is ample space for further optimisation 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 optimises, 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 optimiser, adding another full suite of state of the art optimisations to its stable, this time for the generated JavaScript code.

There exists, however, yet another optimisation niche which is filled by neither of these approaches: optimising 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 optimising 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 optimisations consist mostly of relatively uninteresting cleanup: removing unnecessary assignments, shrinking expressions to smaller or more efficient equivalents, and so on. Some optimisations performed are larger in scope, however, and deserve a more in-depth treatment.

6.1 Tail call elimination

As explained in section 3.2, 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 3.2.

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.

```
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

Figures 2 and 3 give an example of a tail recursive multiplication function before and after this loop transformation respectively.

Loop transformation in the presence of closures However, the loop transformation optimisation 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

```
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

of the mutating variables, instead of capturing the mutable references it would have had without this explicit copying.

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

6.2 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 optimisation, proposed by Loitsch and Serrano [2007], 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 de-trampolined, the amount of actual tail calls eliminated by this optimisation 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 optimisations, 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 optimisation described Schinz and Odersky [2001] to trade stack space for speed.

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 zero, 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, amortising 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 optimisation, where *n* is equal to 1.

6.3 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 Jones [2004]. However, as discussed in section 3.1, 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 optimiser, 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.

7 Performance evaluation and discussion

As indicated in section 1, 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* [Partain, 1993] benchmark suite, to measure Haste's performance compared to the state of the art GHCJS compiler as well as the performance impact made by the optimisations described in section 6. 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 13th 2015, using the `-O2` flag for optimisations. The resulting JavaScript programs were executed using version 4.1.1 of the Node.js interpreter.

7.1 Related work

As the idea of a web-targeting compiler is not new in any sense of the word, there exists a wealth of compilers apart from Haste that attempt to produce efficient JavaScript code from some higher-level functional language. This section gives an overview of the field, comparing Haste to the more prominent web-targeting compilers for functional languages.

Clean The Clean compiler is able to generate JavaScript through the Sapl intermediate language [Bruël and Jansen, 2010]. 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 7.2 of 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.

GHCJS GHCJS [Nazarov et al., 2015] 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 maximise 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 7.

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 [Dijkstra et al., 2013]. As discussed in section 1, UHC fares poorly compared to GHC-based compilers both in terms of performance and in terms of its feature set [Ekblad, 2012]. UHC takes the same approach as GHCJS towards JavaScript compilation, in mainly retargeting its low-level native-targeted output towards JavaScript instead of employing a higher-level translation scheme. This connection, along with GHCJS' position as the de facto web-targeting Haskell compiler and better performance, leads us to use GHCJS rather than UHC as the yardstick with which to measure the Haste compiler.

7.2 Relative performance of the Haste compiler

We measure the performance of Haste as compared to 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 [Nazarov et al., 2015]. 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

Program	Haste	Haste (min.)	GHCJS	Speedup
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

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 negligible impact on execution times, with only the *circsim* benchmark 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.

7.3 Performance of tail call optimisations

The trampolining optimisations described in section 6 – 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 optimisation is the acyclic tail call elimination, which removes unnecessary tail calls and trampolining during a whole program optimisation pass. Turning this optimisation off results in a 10 % slowdown across the entire set of benchmarks.

The utility of the loop transformation optimisation is less universal, but its effect on execution speed can be more pronounced where the optimisation applies. For the *queens* benchmark, disabling the loop transformation optimisation results in a 25 % slowdown. For the *binary-trees* benchmark, disabling this optimisation 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 optimisation, the tail chain counter optimisation 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 optimisation 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 optimisation kicks in it is highly effective: for *binary-trees* disabling the tail chain counter resulted in a 30 % slowdown. For *queens*, disabling the optimisation 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 optimisations, 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 optimisations 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

7.4 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 serialisation of the data constructor and accessor primitives. In order to evaluate the performance of the three different ADT representations discussed in section 4 – 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.

7.5 Limitations

While relying heavily on the web browser's capabilities to enable a shallow translation and lightweight Haskell runtime gives the Haste compiler a considerable performance edge, it is a design choice not without downsides. Chief among them is that features not natively supported by browsers can be harder – or even impossible – to implement. The only practical example of this is *weak references*: references which do not prevent referenced objects from being garbage collected. As Haste relies on the browser for garbage collection and JavaScript does not support weak references, supporting them in Haste is not viable without adding considerable instrumentation code, effectively implementing a garbage collector in addition to the browser's native one.

When presented with a program making use of weak references, Haste will give a compile-time warning and use non-weak references to provide some hobbled, makeshift support for applications using them.

Another memory-related weakness is that by relying on JavaScript's built-in function calls, we limit ourselves to use whichever size of call stack the browser implementor chooses for us. As Haskell programs commonly make use of far deeper recursion than idiomatic JavaScript programs, this can lead to exhaustion of the available call stack in extreme cases. While trampolining mitigates this issue for function calls in tail position, it this does nothing to help with the general case.

While switching to a calling convention using an explicit call stack would solve this issue, this transformation throws any JavaScript optimisation and code analysis involving function calls out the window. Additionally, moving from JavaScript's highly optimised native calling conventions to an implementation in JavaScript itself, is in itself a modification bound to have severe performance penalties. Particularly since this would also involve JavaScript arrays rather than an efficient native stack.

7.6 Conclusions and future work

In this paper 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. Save for the translation scheme itself, none of the techniques used in the Haste compiler are entirely novel in and of themselves. However, the Haste compiler is to our knowledge the first combined application of these techniques, and we take this opportunity to make a more thorough analysis of their performance impact.

As shown in our performance evaluation, our proposed translation scheme, augmented with 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 up to 2.4 and 10 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 we have explored further in our previous work on foreign-function interfaces for web-targeting compilers [Ekblad, 2015].

Recent work on *join points* by Maurer et al. [2017] shows some promise for even greater performance increases, and a study of its impact on web-targeting code remains future work.

8 Bibliography

V. Balat, P. Chambart, and G. Henry. Client-server Web applications with Ocsigen. In *WWW2012*, page 59, Lyon, France, Apr. 2012. URL https://hal.archives-ouvertes.fr/hal-00691710.

N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java Bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 129–140, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: 10.1145/289423.289435.

E. Bruël and J. M. Jansen. Implementing a non-strict purely functional language in JavaScript. *Implementation of Functional Languages*, 2010.

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. doi: 10.1007/978-3-642-41582-1_3.

C. Done. Fay, JavaScript, etc. http://chrisdone.com/posts/fay, 2012.

A. Ekblad. Towards a declarative web. Master's thesis, University of Gothenburg, 2012. Also available from http://ekblad.cc/pubs/hastereport.pdf.

A. Ekblad. Foreign exchange at low, low rates. http://ekblad.cc/ifl15.pdf, 2015.

P. Freeman. Purescript. http://www.purescript.org/, 2016.

G. Guthrie. Your transpiler to JavaScript toolbox. http://luvv.ie/2014/01/21/your-transpiler-to-javascript-toolbox/, 2014.

J. Hughes. Why functional programming matters. *The computer journal*, 32 (2):98–107, 1989.

F. Indutny. The bn.js library. https://github.com/indutny/bn.js.

A. Klein. Chromium source repository. https://chromium.googlesource.com/v8/v8.git/+/1769f892cef0822e6a8b5334e2ad909a0c33e906, 2017.

F. Loitsch and M. Serrano. Hop client-side compilation. *Trends in Functional Programming*, 8:141–158, 2007.

S. Marlow and S. Peyton Jones. The new GHC/Hugs runtime system. *URL http://research. microsoft. com/apps/pubs/default. aspx*, 1998.

S. Marlow and S. Peyton 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. ISBN 1-58113-905-5. doi: 10.1145/1016850.1016856.

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. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291194.

L. Maurer, P. Downen, Z. M. Ariola, and S. Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 482–494. ACM, 2017.

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.

V. Nazarov, H. Mackenzie, and L. Stegeman. GHCJS Haskell to JavaScript compiler. https://github.com/ghcjs/ghcjs, 2015.

W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

T. Petricek and D. Syme. AFAX: Rich client/server web applications in F#. 2007.

S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:

127–202, 1992. ISSN 1469-7653. doi: 10.1017/S0956796800000319. URL http://journals.cambridge.org/article_S0956796800000319.

M. Scheevel. NORMA: a graph reduction processor. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 212–219. ACM, 1986.

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

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.

A. Wirfs-Brock. ECMAScript 2015 language specification. http://www.ecma-international.org/ecma-262/6.0/, 2015.

D. Wirtz. The long.js library. https://github.com/dcodeIO/long.js.

Foreign Exchange at Low, Low Rates

Abstract

We present a novel yet simple foreign function interface, designed for web-targeting Haskell dialects but also applicable to a wider range of high-level target languages. The interface automates marshalling, eliminates boilerplate code, allows increased sanity checking of external data, allows the import of functions as well as arbitrary expressions of JavaScript code, and is implementable as a plain Haskell '98 library without any modification to the Haskell compiler or environment.

We give an implementation of this interface for the JavaScript-targeting Haste compiler, and show how the basic implementation can be further optimised with minimal effort to perform on par with Haskell's vanilla foreign function interface, as well as extended to support automatic marshalling of higher-order functions and automatic marshalling of host language exceptions. We also discuss how the interface may be extended beyond the web domain and implemented across a larger range of host environments and target languages.

1 Introduction

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 [Yakeley, 2014]. 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. When using compilers such as the Haste [Ekblad and Claessen, 2014] and GHCJS [Nazarov et al., 2015] – GHC-based Haskell compilers which target the web browser – the situation is even worse. 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; the user has a wealth of high-level JavaScript libraries within arm's reach, but is forced to go through the low-level gateway of the Haskell FFI [Chakravarty, 2003] to touch them. While the example given in figure 5 certainly works when compiled with either Haste or GHCJS, it is not something the user would like to write.

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

```
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 stdcall unsafe "time.h gettimeofday"
   gettimeofday :: Ptr CTimeval → Ptr () → IO CInt

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

Figure 5: Foreign imports using the vanilla Foreign Function Interface

ambiguity or clash between different languages' built-in representation of various higher-level data structures, as there simply *are* no higher-level data structures on the interface level.

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 produced by compilers such as Haste and GHCJS: said Haskell implementations commonly rely on the browser environment for a large part of their runtime and internally use 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 or 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.

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

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

Figure 6: Foreign imports using our FFI

- Users should be able to easily define their own marshalling schemes for arbitrary types.
- The FFI should allow importing arbitrary snippets of foreign code, not just named, statically known functions. This allows users to efficiently compose code from different libraries in a single import, as well as transform data which may be inefficient or cumbersome to import into Haskell as-is.

  To be clear, this capability is not intended to subsume writing proper, external JavaScript, but to give a means of reducing the impedance mismatch between Haskell and JavaScript without forcing the user to create stub after stub after stub.
- Finally, the FFI should be easy to implement and understand, ideally being implementable without compiler modifications, portable across Haskell implementations targeting high-level 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.

Note that the => symbol here denotes a JavaScript anonymous function. An alternative, more verbose, way to write the foreign code for `getCurrentTime` would be to use the *function* keyword: `function() var ms = ...; return ...;`

Contrasting this example with the standard FFI code from figure 5:

- The low-level machine 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 from the building blocks available to us, without having to resort to external stubs.

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 many 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 6.5, 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.

Our contribution In section 2, we present a novel interface for a web-targeting Haskell dialect to interface with its JavaScript host environment at a high level of abstraction, and describe its implementation for the Haste compiler. The interface lets users import arbitrary JavaScript expressions in addition to the named functions traditionally importable through Haskell's FFI, exploiting JavaScript's built-in lambda
abstraction for parameter interpolation in lieu of heavier and less portable solutions like anti-quotes. This enables users to create efficient bindings to foreign code with a potentially high impedance mismatch without having to pay with excessive boilerplate code. It allows for context dependent sanity checking of incoming data, improving the safety of foreign functions.

The interface makes use of dynamic code evaluation and the fact that JavaScript – the "machine language" of the web – is intended for human consumption to achieve a surprisingly lightweight implementation, which does not rely on modifications to the Haskell compiler; a feat which, to our knowledge, we are the first to perform.

The basic interface is implementable using plain Haskell '98 with the Foreign Function Interface extension, and is extensible by the user in the types of data which can be marshalled as well as in how said marshalling is performed.

In section 3 we discuss various safety and performance concerns about our implementation, and show how these concerns can be alleviated by reaching outside the confines of Haskell '98.

In section 4 we show the flexibility of our design by using it to implement marshalling of higher-order functions between Haskell and JavaScript, as well as a mechanism for automatically marshalling JavaScript exceptions into Haskell equivalents. We also discuss how to remove dynamic code evaluation from the equation with a slight modification to the Haskell compiler in use.

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. Its representation is not fixed, but rather a reference to a value of any type representable in the underlying host language. From the Haskell point of view, its representation can be seen as a completely opaque reference. Hence, the only parts of the library that can interact directly with a `HostAny` value are the ones explicitly imported through the vanilla FFI. A data type is considered to be marshallable if and only if it can be converted to `HostAny` and back again using some such imported function or combination thereof.

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.

```
type HostAny

class ToAny a where
  toAny :: a → HostAny

class FromAny a where
  fromAny :: 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 ToAny Int
instance FromAny 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

At first glance, it might seem strange to separate `ToAny` and `FromAny` instead of merging them into a single `Marshal` class. The reason for this is that merging the two classes breaks marshalling of pure higher-order functions in a rather subtle way, as discussed in section 4.

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 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 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.

It is important to note that the Haskell type given to an imported function is assumed to correctly describe its type signature; an assumption which is not – *can* not – be statically checked. Just like casting a C function to an invalid type before calling it through a conventional FFI, attempting to call a function imported through our FFI with a bogus type results in undefined runtime behaviour.

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 requiring some extensions to Haskell '98 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 base 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 stdcall intToAny :: Int → HostAny
foreign import stdcall anyToInt :: HostAny → IO Int

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

The JavaScript implementation of these two functions is simply the identity function. As explained in section 2, `HostAny` is simple an opaque reference to Haskell, but to JavaScript a reference is just another dynamically typed value. These functions are essentially a slightly roundabout way to coerce, rather than convert, base type values into `HostAny` without having to know anything about the compiler's FFI implementation or internal representation of the base types. The same approach is used for the other base types.

We might also find a `HostAny` instance for `ToAny` and `FromAny` handy. Of course, `HostAny` already being in its JavaScript 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?

JavaScript, supports two basic types, which are sufficient to represent values of any non-arrow type: arrays and dictionaries.

Converting Haskell lists into arrays is a relatively straightforward affair. We need two functions: one to create a new, empty array, and one to push a new value onto the end of the array.[1] Converting arrays back into lists is similarly easy: we simply need to obtain the array's length, and read the requisite number of elements back into Haskell, building a list as we go along.

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 Haskell representations are quite similar, or even identical. Hence, we need to put the power over this decision into the hands of the user, providing functionality to build as well as inspect 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 sum and product types to and from dictionaries: `mkDict` which creates dictionaries from association lists, and `getMember`, which looks up dictionary values by key. While a `Map` would normally be the go-to data structure for describing dictionaries, `mkDict` only iterates over its argument in order to add the corresponding entries to the created JavaScript dictionary, making simple association lists a less heavyweight choice than a `Map`.

The complete implementation of marshalling for lists and dictionaries is shown in figure 8.

[1] A JavaScript array is quite a different beast from an "actual" array as seen in C, making the push operation more efficient than one would normally expect.

```
foreign import stdcall newDict :: IO HostAny
foreign import stdcall newArr :: IO HostAny

foreign import stdcall set :: HostAny → HostString → HostAny → IO ()
foreign import stdcall get :: HostAny → HostString → IO HostAny
foreign import stdcall push :: HostAny → HostAny → IO ()

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

instance ToAny a ⇒ ToAny [a] where
  toAny xs = unsafePerformIO $ do
    arr ← newArray
    mapM_ (push arr . toAny) xs
    return arr

instance FromAny a ⇒ FromAny [a] where
  fromAny arr = do
    len ← fromAny =<< get arr (toHostString "length")
    sequence
      [ fromAny =<< get arr (toAny i)
      | i ← [0..len-1 :: Int]
      ]

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

Figure 8: Marshalling arrays and dictionaries

Note the use of the generally unsafe `unsafePerformIO` in `mkDict` and in the `ToAny` instance for lists. The only side effects performed by said functions are to create a new references, mutate them, and then return them, never to mutate them again. As the references to the mutated objects are not accessible outside said functions until after all mutation has taken place, these side effects are not observable and this use of `unsafePerformIO` can thus be considered safe.

Together with the previously defined instances for base types, this gives us the power to marshal any non-arrow data type into an equivalent `HostAny` value and back again. 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

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

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

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 the `host` function – the function by which JavaScript functions are imported into Haskell – turns out to be slightly trickier than marshalling data between environments. We want to be able to use a single function to import any JavaScript function, using the declared Haskell type of the imported function to determine its arity, argument types and return type. There is a well known way to accomplish this, colloquially known as "the printf trick" [Augustsson and Massey, 2013], which uses an inductive class instance to successively build up a list of arguments over repeated function applications, and a base case instance to perform some computation over said arguments after the function in question has been fully applied. 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.

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

foreign import stdcall apply :: HostFun → HostAny → IO HostAny

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

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

When applied to some `HostFun` and a list of arguments collected so far, `import_` returns a variadic function f, whose arity is decided by how many arguments it is applied to or by explicit type annotation. When f is applied to an argument, said argument is marshalled into a `HostAny` value and added to the list of arguments. When f is fully applied and we reach the base case – a nullary computation in the `IO` monad – the `HostFun` provided to `import_` is shipped off to JavaScript via the vanilla FFI to be applied to the list of arguments built up during the recursion. After `apply` returns, its return value is marshalled back into Haskell through `fromAny` and returned to the caller of f. The `apply` function which performs the actual application is very simple:

```javascript
function(f, args) {
  return f.apply(null, 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`.

```haskell
foreign import stdcall 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. `eval` is then used to create a function object – represented as a `HostFun` – which is used to create the aforementioned Haskell function f. This is all we need to be able to import first order JavaScript functions such as the motivating example in figure 6

3 Optimising 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 optimise marshalling, or even completely unroll and eliminate some of the basic interface's primitive operations, for instance – there are several general optimisations 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 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* [Peyton Jones et al., 2001], we can eliminate this pointless work in most cases by specialising 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 optimise function application for. The actual specialisation 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 optimisation, 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!

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

instance FromAny a ⇒ Import (IO a) where
  import_ = dispatch

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

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

Figure 10: Specialising the `host` base case

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, on which both Haste and GHCJS are based, gives us the means to eliminate this potential pitfall. The *StaticPointers* extension, its first incarnation described in [Epstein et al., 2011], 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:

```haskell
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.[2] 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 optimise 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 optimiser with focus on code size – before deployment. Not only do such optimises suffer the same analytical difficulties as the language runtime itself from the presence of dynamically 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 Extending our interface

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 higher

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

order functions, exception handling and generic marshalling. In this section we demonstrate the flexibility of our interface by showing how this functionality can be implemented 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.

```haskell
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.

```haskell
foreign import stdcall 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: StablePointers [Reid, 1994]. 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 6.3, and an alternative solution is presented.

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

foreign import stdcall _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 [Eisenberg et al., 2014], to lend us the type level flexibility we need. We begin by defining the Exportable type class which denotes all functions that can be exported into JavaScript, and a closed type family describing the type level behaviour of our function marshalling.

```haskell
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, 1994], 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 6.3 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. This is the reason for having two separate type classes for marshalling data into and out of Haskell. We need to be able to *export* pure functions from Haskell while for safety reasons not allowing them to be *imported*, and we want to avoid creating the problematic unlimited export instance described above; forcing importable types to be exportable and vice versa disallows both.

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 necessary: the open type families originally introduced by Chakravarty et al [Chakravarty et al., 2005] do not admit the overlapping type equations required to make pure functions an instance of `Exportable`.

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

foreign import stdcall _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

class Exportable f where
  mkHostFun :: f → Host f

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

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 – the vanilla FFI's mechanism for making Haskell functions available to foreign code. 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 :: ToAny 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. As stated in section 2.2, any non-arrow Haskell type can be represented using a combination of arrays and dictionaries. Using one of the generic programming frameworks offered by Haskell, such as GHC generics [Magalhães et al., 2010] or Template Haskell [Sheard and Jones, 2002], it is possible to create a *default instance* of the marshalling type classes, applicable to any Haskell type.

As the implementation of such an instance is neither novel nor particularly interesting in the context of this paper, we refer the reader to the one used by the *aeson* package for encoding and decoding of JSON values [O'Sullivan, 2015]. This default instance provides the final piece of the puzzle required to use the interface as presented in figure 6.

4.4 Marshalling JavaScript exceptions

Trapping errors in foreign C code is relatively straightforward, albeit cumbersome, owing to the relative absence of structured error handling in C. However, when interacting with a higher-level language, one must take into account the risk of exceptions being raised in any imported foreign code. In the basic Haste.Foreign interface, such exceptions must be manually handled lest they terminate the enclosing Haskell program much like a segmentation fault in imported C code would terminate a native Haskell program.

Thankfully, we can leverage the higher-order import capabilities described in section 4 to catch JavaScript exceptions and re-throw them within Haskell's exception handling framework. We import a JavaScript function `catchJS` which accepts an exception handler function and an IO computation as its arguments. When called, `catchJS` executes IO computation inside a try-catch block. If an exception is raised, it is passed to the exception handler function which then takes appropriate action.

```
catchJS :: (ToAny a, FromAny a)
        ⇒ (String → IO ())
        → IO a → IO a
catchJS = "(handle, act) ⇒\
  \{ try       { return act(); }\
  \  catch (ex) { handle(ex.toString()); }}"

data HostException = HostException String
  deriving Show
instance Exception HostException

class Safely a where
  safely :: a → a

instance Safely b ⇒ Safely (a → b) where
  safely f x = safely (f x)

instance (FromAny a, ToAny a) ⇒ Safely (IO a) where
  safely m = catchJS (throwIO . HostException)

very_safe_host :: Import a ⇒ StaticPtr String → a
very_safe_host = safely safe_host
```

Figure 12: Marshalling JavaScript exceptions

`catchJS` may be used similarly to Haskell's `catch` function to catch exceptions in foreign functions where they are expected to occur. However, it is not necessarily the case that we always want to handle exceptions right at the call site of a foreign function – quite the opposite! Instead, we can create an exception-safe equivalent to `host` which uses `catchJS` to dispatch *all* calls to functions imported through it, with an exception handler function that simply re-throws the JavaScript exception wrapped in a Haskell exception. The wrapped exception can then be caught anywhere a "normal" Haskell exception could be caught. A complete implementation of this extension is given in figure 12.

An interesting side-effect of this approach to exception handling and the fact that JavaScript syntax errors are catchable exceptions is that the exception-safe `host` function is not only catch dynamic errors, but incorrectly written foreign imports as well.

It should be noted that this approach incurs a performance penalty due to the extra function call and marshalling required to dispatch a function, as further discussed in section 5.

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.
- *HOF 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 applied 500 000 times 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 containing 500 000 elements, to obtain higher-level code which is harder to optimise and analyse for strictness.

The resulting programs, compiled with version 0.5.4.2 of the Haste compiler which incorporates all the optimisations described in section 3,

	Tight loop	mapM_	Tight + exceptions
Outbound	0.98	1.07	8.21
In-out	0.97	1.08	11.70
Product types	0.83	0.95	2.42
HOF import	0.94	0.96	1.09

Table 3.1: Execution times as fractions of FFI execution times

were then repeatedly executed using version 4.2.2 of 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. The benchmarks were executed on a Lenovo ThinkPad X230 laptop running Debian GNU/Linux, equipped with an Intel i5 3210M CPU and 8 GB of RAM.

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 both the highly optimised and less optimised loop cases, with the loose loop showing a modest 7 % slowdown over the vanilla FFI, and the tight loop even eking out a tiny performance benefit.

In contrast, the performance hit when using the exception-safe version defined in section 4.4 is huge. This is by no means surprising: using the exception-safe version entails marshalling no less than two additional higher-order functions, which is quite a bit heavier than the otherwise very lightweight marshalling performed for plain numbers.

In-out Moving on to the benchmarks where we actually marshal incoming data, the picture is much the same as for the *outbound* benchmark. The performance hit from the exception marshaller becomes even more problematic here, as the return value of the function needs to be marshalled first into Haskell, then back into JavaScript for the exception handler, and finally back into Haskell again.

Product types Our interface shows a small performance advantage when it comes to marshalling more complex values, being 5 – 20 % faster depending on the loop. Our assumption about peeking and poking at pointers being suboptimal in an environment where such operations are considerably more expensive than on bare metal seems to have been correct.

Worth noting is that the performance overhead of the exception marshaller becomes significantly less prohibitive in this benchmark, as the complexity of marshalling grows. This indicates that while expensive compared to the minimal marshalling required for base types, exception safety may not be all that expensive after all, when calling a function which performs actual work.

HOF import Our interface seems to compare favourably to the vanilla FFI for this case, although the performance gain difference is quite minimal. This is to be expected, as the heavy lifting required to export Haskell functions into JavaScript is relatively similar and quite heavy regardless of how the relatively lightweight marshalling of the function's base type arguments is carried out. Again, it is worth pointing out how the significance of the exception marshaller's performance penalty dwindles as the marshalling process as a whole grows heavier.

Performance verdict: acceptable Judging by these numbers the performance of our library is quite acceptable, with the exception of the heavy toll taken by the exception marshaller on the less complex marshalling cases. Interestingly, the optimisation described in section 3.3 does not impact performance measurably in these benchmarks. If any performance benefit is to be had from this optimisation, is will likely come from increased opportunities for minifiers and JavaScript engines to perform inlining over more complex programs.

It is encouraging that our interface's intended use case - marshalling more complex types and higher-order functions - 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, or at least handling JavaScript exceptions in foreign code instead of relying on our library's exception marshaller, may be an attractive option to reduce the performance penalty incurred by our interface in unfavourable circumstances, 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 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 and error handling

The `fromAny` function used to implement marshalling in section 2 is by definition not total. As its purpose is to convert dynamically typed JavaScript values into statically 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.

It should also be noted that the basic interface does neither handles syntax errors nor exceptions thrown from foreign code: it is the responsibility of the user

6.2 Generalising to other languages

The implementation so far has been quite clearly focused on the needs of web-targeting Haskell dialects. However, the interface and library described should be portable across other host languages with relative ease, provided that they have the following properties:

Dynamically typed. The reliance on the `HostAny` type to represent host language values makes support for statically typed host languages very cumbersome, at best.

Garbage collected. Our interface completely ignores something which very much concerns traditional foreign function interfaces: ownership and eventual deallocation of memory. This careless behaviour is enabled by the fact that the host language is assumed to be garbage collected. Removing garbage collection from the equation would land us in a position much more similar to the vanilla FFI, albeit with less restrictions on marshallable types.

Dynamic code evaluation. The ability to import foreign code without compiler modification relies crucially on the ability of the host language to execute arbitrary strings of code. While this restriction can be lifted with a compiler modification as described in 3.3, this diminishes the utility of the interface, limiting the portability and ease of implementation that are some of its greatest strengths.

Higher order. Dynamic code evaluation is not much use if we can't evaluate a piece of code and get a function object back – there is nothing for our Haskell program to call! While it is certainly conceivable to re-evaluate strings of host language code anew on each function application, with arguments spliced into the evaluated program, this strikes us as an approach which is both brittle and slow, leading us to conclude that our interface would fare quite badly in a first order language.

Languages that fulfil these criteria and might make attractive targets for porting include Python, PHP and Ruby. While our interface should be portable to any such language, we have yet to implement our interface for any non-JavaScript environment.

6.3 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 collection having to deal with low level details such as memory management is the core motivation for this work, rectifying this issue 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 is 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.

```haskell
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:

```haskell
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.

6.4 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.

6.5 Related work

Aside from the vanilla foreign function interface used as the basis of our interface, there are several different, more modern, takes on interfacing purely functional languages with host language code. One common denominator is specialisation: without exception, these implementations rely in large part on modifications to the compiler or language itself, in contrast to our interface which makes some sacrifices in order to be implementable as a library, maximising portability across host and Haskell implementations alike.

Idris: host-parametric FFI *Idris* is a dependently typed, Haskell-like language with backends for several host environments,
JavaScript being one of them [Brady, 2015]. 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, and parameterised 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 parameterised 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.

Fay: featureful but static Our interface was partially inspired by the foreign function interface of the *Fay* language, a "proper subset of Haskell that compiles to JavaScript" [Done, 2015]. While the two are very similar in syntax, allowing users to import typed strings of host language code, Fay's solution is highly specialised. 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: JavaScriptFFI The *GHCJS* Haskell-to-JavaScript compiler [Nazarov et al., 2015] utilises 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 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.

UHC: the traditional FFI, on steroids for JavaScript The UHC Haskell compiler comes with a JavaScript backend as well, and matching higher-level extensions to its foreign function interface [Dijkstra et al., 2012]. 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.

Clean: mixing host and client language code The Clean language sports a foreign function interface which differs slightly from the rest of the interfaces discussed 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 [Plasmeijer and van Eekelen, 2002]. 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, allowing host and client language code to be mixed, but less so than the other interfaces discussed in this section due to its less expedient marshalling capabilities.

Quasiquoting Quasiquoting represent another, more radically different, approach to the problem of bridging with a host language [Mainland, 2007]. 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 [Chakravarty, 2014].

This usefulness comes at the price of a more involved implementation. Quasiquoting requires explicit compiler support in the form of compile time template metaprogramming as well as special extensions for running the quasiquoters 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.

7 Conclusions and future work

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

As described in section 6.2 our interface has yet to be implemented for host languages other than JavaScript. Demonstrating that it is practically portable to at least one other host language would give additional weight to our claims of portability and improve the general applicability of the interface.

By combining two optimisations given in section 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 remove the requirement 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, nearly making our interface implementable on recent versions of the Java Virtual Machine if not for the dynamic typing requirement. Investigating ways around this restriction and an implementation of our interface for the Java Virtual Machine, with the prerequisite Haskell-to-JVM compiler, would lend additional applicability to our interface.

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.

Conclusions We have presented the design and implementation of a novel, portable foreign function interface for web-targeting Haskell dialects. While designed for the web sphere, the given implementation is also applicable to a wide range of other high level target languages as well.

We have also given a number of optimisations, improving the performance and safety of our interface and lightening the restrictions placed on the host environment, and implemented our interface as a library for the Haste Haskell-to-JavaScript compiler. Finally, we have used this library to further extend our marshalling capabilities to cover functions and foreign exceptions, 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: web-targeting Haskell implementations.

8 Acknowledgements

This work has been partially funded by the Swedish Foundation for Strategic Research, under grant RAWFP. Many thanks to Koen Claessen, Emil Axelsson and Atze van der Ploeg for their valuable feedback and comments.

9 Bibliography

L. Augustsson and B. Massey. The *Text.Printf* module. http://hackage.haskell.org/package/base-4.8.0.0/docs/Text-Printf.html, 2013.

E. Brady. Cross-platform compilers for functional languages. *Under consideration for Trends in Functional Programming*, 1, 2015.

M. M. Chakravarty. *The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*. 2003.

M. M. Chakravarty. Foreign inline code: systems demonstration. In *ACM SIGPLAN Notices*, volume 49, pages 119–120. ACM, 2014.

M. M. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ACM SIGPLAN Notices*, volume 40, pages 241–253. ACM, 2005.

A. Dijkstra, J. Stutterheim, A. Vermeulen, and S. D. Swierstra. Building javascript applications with haskell. In *Implementation and Application of Functional Languages*, pages 37–52. Springer, 2012.

C. Done. Fay programming language. https://github.com/faylang/fay/wiki, 2015.

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. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535856. URL http://doi.acm.org/10.1145/2535838.2535856.

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. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633367. URL http://doi.acm.org/10.1145/2633357.2633367.

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. ISBN 978-1-4503-0860-1. doi: 10.1145/2034675.2034690. URL http://doi.acm.org/10.1145/2034675.2034690.

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. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863529. URL http://doi.acm.org/10.1145/1863523.1863529.

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. ISBN 978-1-59593-674-5. doi: 10.1145/1291201.1291211. URL http://doi.acm.org/10.1145/1291201.1291211.

V. Nazarov, H. Mackenzie, and L. Stegeman. GHCJS Haskell to JavaScript compiler. https://github.com/ghcjs/ghcjs, 2015.

B. O'Sullivan. The *aeson* package. http://hackage.haskell.org/package/aeson-0.11.1.0/, 2015.

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.

R. Plasmeijer and M. van Eekelen. Clean language report version 2.1, 2002.

A. Reid. Malloc pointers and stable pointers: Improving Haskell's foreign language interface. In *Glasgow Functional Programming Workshop Draft Proceedings, Ayr, Scotland*. Citeseer, 1994.

T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

A. Yakeley. The *time* package. http://hackage.haskell.org/package/time, 2014.

Paper III

A Seamless, Client-Centric Programming Model for Type-Safe Web Applications

Abstract

We propose a new programming model for web applications which is (1) seamless; one program and one language is used to produce code for both client and server, (2) client-centric; the programmer takes the viewpoint of the client that runs code on the server rather than the other way around, (3) functional and type-safe, and (4) portable; everything is implemented as a Haskell library that implicitly takes care of all networking code. Our aim is to improve the painful and error-prone experience of today's standard development methods, in which clients and servers are coded in different languages and communicate with each other using ad hoc protocols. We present the design of our library called Haste.App, an example web application that uses it, and discuss the implementation and the compiler technology on which it depends.

1 Introduction

Development of web applications is no task for the faint of heart. The conventional method involves splitting your program into two logical parts, writing the one in JavaScript, which is notorious even among its proponents for being wonky and error-prone, and the other in any compiled or server-interpreted language. Then, the two are glued together using whichever home-grown network protocol seems to fit the application. However, most web applications are conceptually single entities, making this forced split an undesirable hindrance which introduces new possibilities for defects, adds development overhead and prevents code reuse.

Several solutions to this problem have been proposed, as discussed in section 5.1, but the perfect one has yet to be found. In this paper, we propose a functional programming model in which a web application is written as a single program from which client and server executables are generated during compilation. Type annotations in the source program control which parts are executed on the server and which are executed on the client, and the two communicate using type safe RPC calls. Functions which are not explicitly declared as server side or client side are usable by either side.

Recent advances in compiler technology from functional languages to JavaScript have led to a wealth of compilers targeting the web space, and have enabled the practical development of functional libraries and applications for the browser. This enables us to implement our solution as a simple Haskell library for any compiler capable of producing JavaScript output, requiring no further modification to existing compilers.

As our implementation targets the Haste Haskell to JavaScript compiler [Ekblad , 2012], this paper also goes into some detail about its design and implementation as well as the alternatives available for compiling functional languages to a browser environment.

Motivation Code written in JavaScript, the only widely supported language for client side web applications, is often confusing and error-prone, much due to the language's lack of modularity, encapsulation facilities and type safety.

Worse, most web applications, being intended to facilitate communication, data storage and other tasks involving some centralised resource, also require a significant server component. This component is usually implemented as a completely separate program, and communicates with the client code 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 [Lubbers, Greco , 2010], we may come up with something like the program in figure 13 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

```
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 13: JavaScript chatbox implementation

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.

Our proposed model, dubbed "Haste.App", has 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 specialised 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 serialisable through the `Serialize` type class, and produces a typed identifier which may be used to refer to the remote function. In the *Hello World* example, the type of `greetings` is `Remote (String → Server ())`, indicates 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 serialised and sent over a network connection. When a `Remote` function is applied to an argument using `<.>`, the argument is serialised 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

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

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

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

Figure 14: server side state: doing it properly

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 14 shows how to make proper use of server side state.

2.3 The chatbox, revisited

Now that we have seen how to implement both network communication, 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;
- 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 $ \cs →
      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"] $ \[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.

```
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 15: Types of the Haste.App core functions

```
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.

```
msgbox ‘onEvent‘ OnKeyPress $ \13 → 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 15.

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 serialisable 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

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 serialised 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 handling clutter in the interest of clarity. The complete implementation is available for download, together with the Haste compiler, from Hackage as well as from our website at `http://haste-lang.org`.

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 behaviour 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 initialised 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.

Client side implementations 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 serialised 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 optimised 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 = \_ → fmap toJSON m

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

In essence, any function, over any number of arguments, which returns a serialisable 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. [Taylor , 2013]

The final function operating on these types is `liftServerIO`, used to initialise 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.

Server side implementations 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. The `liftServerIO` function 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 [Epstein et al , 2011], which is discussed further in section 5.1.

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* [Claessen , 1999]. 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. [Peyton Jones , 2001] This concurrency-enhanced IO monad is used as the basis of the `Client` monad.

```haskell
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`.

```haskell
newResult :: Client (Nonce, MVar JSON)
newResult = do
  (nonce, m) ← get
  mv ← 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, mv) ← newResult
  webSocketSend $
    toJSON (nonce, identifier, reverse args)
  fromJSON <$> takeMVar mv
```

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 The Haste compiler

In order to allow the same language to be used on both client and server, we need some way to compile that language into JavaScript. To this end, we make use of the Haste compiler [Ekblad , 2012], started as an MSc thesis and continued as part of this work. Haste builds on the GHC compiler to provide the full Haskell language, including most GHC-specific extensions, in the browser.

As Haste has not been published elsewhere, we describe here some key elements of its design and implementation which are pertinent to this work.

4.1 Choosing a compiler

Haste is by no means the only JavaScript-targeting compiler for a purely functional language. In particular, the GHC-based GHCJS [Nazarov , 2012] and UHC [Dijkstra et al , 2013] compilers are both capable of compiling standard Haskell into JavaScript; the Fay [Done , 2012] language was designed from the ground up to target the web space using a subset of Haskell; and there exist solutions for compiling Erlang [Guthrie , 2014] and Clean [Domoszlai et al , 2011] to JavaScript as well. While the aforementioned compilers are the ones most interesting for purely functional programming, there exist a wealth of other JavaScript-targeting compilers, for virtually any language.

Essentially, our approach is portable to any language or compiler with the following properties:

- The language must provide a static type system, since one of our primary concerns is to reduce defect rates through static typing of the client-server communication channel.
- The language must be compilable to both JavaScript and a format suitable for server side execution as we want our web applications to be written and compiled as a single program.

- We want the language to provide decent support for a monadic programming style, as our abstractions for cooperative multitasking and synchronous client-server communication are neatly expressible in this style.

As several of the aforementioned compilers fulfil these criteria, the choice between them becomes almost arbitrary. Indeed, as Haste.App is compiler agnostic, this decision boils down to one's personal preference. We chose to base our solution on Haste as we, by virtue of its authorship, have an intimate knowledge of its internal workings, strengths and weaknesses. Without doubt, others may see many reasons to make a different choice.

4.2 Implementation overview

Haste offloads much of the heavy lifting of compilation - parsing, type checking, intermediate code generation and many optimisations - onto GHC, and takes over code generation after the STG generation step, at the very end of the compilation process. STG [Peyton Jones , 1992] is the last intermediate representation used by GHC before the final code generation takes place and has several benefits for use as Haste's source language:

- STG is still a functional intermediate representation, based on the lambda calculus. When generating code for a high level target language such as JavaScript, where functions are first class objects, this allows for a higher level translation than when doing traditional compilation to lower level targets like stack machines or register machines. This in turn allows us to make more efficient use of the target language's runtime, leading to smaller, faster code.

- In contrast to Haskell itself and GHC's intermediate Core language, STG represents 'thunks', the construct used by GHC to implement non-strict evaluation, as closures which are explicitly created and evaluated. Closures are decorated with a wealth of information, such as their set of captured variables, any type information needed for code generation, and so on. While extracting this information manually is not very hard, having this done for us means we can get away with a simpler compilation pipeline.

- The language is very small, essentially only comprising lambda abstraction and application, plus primitive operations and facilities for calling out to other languages. Again, this allows the Haste compiler to be a very simple thing indeed.

- Any extensions to the Haskell language implemented by GHC will already have been translated into this very simple intermediate format, allowing us to support basically any extension GHC supports without effort.

- Application of external functions is always saturated, as is application of most other functions. This allows for compiling most function applications into simple JavaScript function calls, limiting the use of the slower dynamic techniques required to handle curried functions in the general case [Marlow, Peyton Jones , 2004] to cases where it is simply not possible to statically determine the arity of a function.

In light of its heavy reliance on STG, it may be more correct to categorise Haste as an STG compiler rather than a Haskell compiler.

4.3 Data representation

The runtime data representation of Haste programs is kept as close to regular JavaScript programs as possible. The numeric types are represented using the JavaScript `Number` type, which is defined as the IEEE754 double precision floating point type. This adds some overhead to operations on integers as overflow and non-integer divisions must be handled. However, this is common practice in hand-written JavaScript as well, and is generally handled efficiently by JavaScript engines.

Values of non-primitive data types in Haskell consist of a data constructor and zero or more arguments. In Haste, these values are represented using arrays, with the first element representing the data constructor and the following values representing its arguments. For instance, the value `42 :: Int` is represented as `[0, 42]`, the leading `0` representing the zeroth constructor of the `Int` type and the `42` representing the "machine" integer. It may seem strange that a limited precision integer is represented using one level of indirection rather than as a simple number, but recall that the `Int` type is defined by GHC as `data Int = I# Int#` where `Int#` is the primitive type for machine integers.

Functions are represented as plain JavaScript functions, one of the blessings of targeting a high level language, and application can therefore be implemented as its JavaScript counterpart in most cases. In the general case, however, functions may be curried. For such cases where the arity of an applied function can not be determined statically, application is implemented using the eval/apply method described in [Marlow, Peyton Jones , 2004] instead.

```haskell
import Haste.Foreign

-- A MutableVar is completely opaque to Haskell code
-- and is only ever manipulated in JavaScript. Thus,
-- we use the Unpacked type to represent it,
-- indicating a completely opaque value.
newtype MutableVar a = MV Unpacked

instance Marshal (MutableVar a) where
  pack         = MV
  unpack (MV x) = x

newMutable :: Marshal a ⇒ a → IO (MutableVar a)
newMutable = ffi "(function(x) {return {val: x};})"

setMutable :: Marshal a ⇒ MutableVar a → a → IO ()
setMutable = ffi "(function(m, x) {m.val = x;})"

getMutable :: Marshal a ⇒ MutableVar a → IO a
getMutable = ffi "(function(m) {return m.val;})"
```

Figure 16: Mutable variables with `Haste.Foreign`

4.4 Interfacing with JavaScript

While Haste supports the Foreign Function Interface inherited from GHC, with its usual features and limitations [Peyton Jones , 2001], it is often impractical to work within the confines of an interface designed for communication on a very low level. For this reason Haste sports its own method for interacting with JavaScript as well, which allows the programmer to pass any value back and forth between Haskell and JavaScript, as long as she can come up with a way to translate this value between its Haskell and JavaScript representations. Not performing any translation at all is also a valid "translation", which allows Haskell code to store any JavaScript value for later retrieval without inspecting it and vice versa. The example given in figure 16 implements mutable variables using this custom JavaScript interface.

The core of this interface consists of the `ffi` function, which allows the programmer to create a Haskell function from arbitrary JavaScript code. This function exploits JavaScript's ability to parse and execute arbitrary strings at run time using the `eval` function, coupled with the fact that functions in Haste and in JavaScript share the same representation, to dynamically create a function object at runtime. The `ffi` function is typed using the same method as the `mkRemote` function described in section 3. When applied to one or more arguments instantiating the `Marshal` type class, the `pack` function is applied to each argument, marshalling them into their respective JavaScript representations, before they are passed to the dynamically created function. When that function returns, the inverse `unpack` function is applied to its return value before it is passed back into the Haskell world.

As the marshalling functions chosen for each argument and the foreign function's return value depends on its type, the programmer must explicitly specify the type of each function imported using `ffi`; in this, Haste's custom method is no different from the conventional FFI.

There are several benefits to this method, the most prominent being that new marshallable types can be added by simply instantiating a type class. Thanks to the lazy evaluation employed by Haste, each foreign function object is only created once and then cached; any further calls to the same (Haskell) function will reuse the cached function object. Implementation-wise, this method is also very non-intrusive, requiring only the use of the normal FFI to import JavaScript's `eval` function; no modification of the compiler is needed.

5 Discussion and related work

5.1 Related work

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, our solution 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 [Conductance application server , 2014] 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 serialised and de-serialised, 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 [The Opa framework for JavaScript , 2014], 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 optimise.

Google Web Toolkit Google Web Toolkit [Wargolet , 2011], 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 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.

Duetto Duetto [Pignotti , 2013] is a C++ compiler targeting the web, written from the ground up to produce code for both client and server simultaneously. It utilises the new attributes mechanism introduced in C++11 [Stroustrup , 2013] 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. Duetto 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, Duetto 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 Duetto can be initiated implicitly apply to Duetto as well.

Sunroof In contrast to Conductance and Duetto, Sunroof [Bracker, Gill , 2014] 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 [Balat , 2006] 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 [Petricek, Syme , 2007], 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. [Serrano et al , 2006][Cooper et al , 1999][Chlipala , 2010] As our solution aims to bring typed, seamless communication into the existing Haskell ecosystem without language modifications, these languages solve a different set of problems.

Advantages of our approach We believe that our approach has a number of distinct advantages to the aforementioned attacks on the problem.

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 serialisation and de-serialisation 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 prevents 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 tangle with complex compiler internals.

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 [Epstein et al , 2011], 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, making it impossible to call non-existent remote functions and break the program's type safety in other ways.

Another approach to this problem is defunctionalisation [Danvy, Nielsen , 2001], 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 defunctionalised 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 analysing code for improper behaviour is certainly possible, designing and enforcing a security policy sufficiently strict to ensure correct behaviour while flexible enough to be practically useful would be an unwelcome burden on the programmer.

5.2 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 Duetto 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.

6 Future work

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 [Hedin et al , 2014].

While this does indeed provide an effective defence 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.

Real world applications As Haste.App is quite new and experimental, 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 and a more featureful variant on the chatbox example given in section 2.3, 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.

7 Conclusion

We have presented a programming model which improves on the current state of the art in client-server web application development. In particular, our solution combines type safe communication between the client and the server with functional semantics, clear demarcations 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.

Acknowledgements

This work has been partially funded by the Swedish Foundation for Strategic Research, under grant RAWFP.

8 Bibliography

V. Balat. "Ocsigen: typing web interaction with objective Caml." Proceedings of the 2006 workshop on ML. ACM, 2006.

J. Bracker and A. Gill. "Sunroof: A Monadic DSL for Generating JavaScript." In Practical Aspects of Declarative Languages, pp. 65-80. Springer International Publishing, 2014.

A. Chlipala. "Ur: statically-typed metaprogramming with type-level record computation." ACM Sigplan Notices. Vol. 45. No. 6. ACM, 2010.

K. Claessen. "Functional Pearls: A poor man's concurrency monad." Journal of Functional Programming 9 (1999): 313-324.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In Formal Methods for Components and Objects (pp. 266-296). Springer Berlin Heidelberg, 2007.

The Conductance application server. Retrieved March 1, 2014, from http://conductance.io.

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, pp. 162-174. ACM, 2001.

A. Dijkstra, J. Stutterheim, A. Vermeulen, and S. D. Swierstra. "Building JavaScript applications with Haskell." In Implementation and Application of Functional Languages, pp. 37-52. Springer Berlin Heidelberg, 2013.

L. Domoszlai, E. Bruël, and J. M. Jansen. "Implementing a non-strict purely functional language in JavaScript." Acta Universitatis Sapientiae 3 (2011): 76-98.

C. Done. (2012, September 15). "Fay, JavaScript, etc.", Retrieved March 1, 2014, from http://chrisdone.com/posts/fay.

A. Ekblad. "Towards a declarative web." Master of Science Thesis, University of Gothenburg (2012).

J. Epstein, A. P. Black, and S. Peyton-Jones. "Towards Haskell in the cloud." In ACM SIGPLAN Notices, vol. 46, no. 12, pp. 118-129. ACM, 2011.

G. Guthrie. (2014, January 1). "Your transpiler to JavaScript toolbox". Retrieved March 1, 2014, from http://luvv.ie/2014/01/21/your-transpiler-to-javascript-toolbox/.

D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. "JSFlow: Tracking information flow in JavaScript and its APIs." In Proc. 29th ACM Symposium on Applied Computing. 2014.

P. Lubbers and F. Greco. "Html5 web sockets: A quantum leap in scalability for the web." SOA World Magazine (2010).

S. Marlow, and S. Peyton Jones. "Making a fast curry: push/enter vs. eval/apply for higher-order languages." In ACM SIGPLAN Notices, vol. 39, no. 9, pp. 4-15. ACM, 2004.

V. Nazarov. "GHCJS Haskell to JavaScript Compiler". Retrieved March 1, 2014, from https://github.com/ghcjs/ghcjs.

The Opa framework for JavaScript. Retrieved May 2, 2014, from http://opalang.org.

T. Petricek, and Don Syme. "AFAX: Rich client/server web applications in F#." (2007).

S. Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine." J. Funct. Program. 2, no. 2 (1992): 127-202.

S. Peyton Jones. "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell." Engineering theories of software construction 180 (2001): 47-96.

A. Pignotti. (2013, October 31). "Duetto: a C++ compiler for the Web going beyond emscripten and node.js". Retrieved March 1, 2014, from http://leaningtech.com/duetto/blog/2013/10/31/Duetto-Released/.

M. Serrano, E. Gallesio, and F. Loitsch. "Hop: a language for programming the web 2. 0." OOPSLA Companion. 2006.

B. Stroustrup. (2014, January 21). "C++11 - the new ISO C++ standard." Retrieved March 1, 2014, from http://www.stroustrup.com/C++11FAQ.html.

C. Taylor. (2013, March 1). "Polyvariadic Functions and Printf". Retrieved March 1, 2014, from http://chris-taylor.github.io/blog/2013/03/01/how-haskell-printf-works/.

S. Wargolet. "Google Web Toolkit. Technical report 12." University of Wisconsin-Platterville Department of Computer Science and Software Engineering, 2011.

Paper IV

A Meta-EDSL for Distributed Web Applications

Abstract

We present a domain-specific language for constructing and configuring web applications distributed across any number of networked, heterogeneous systems. Our language is embedded in Haskell, provides a common framework for integrating components written in third-party EDSLs, and enables type-safe, access-controlled communication between nodes, as well as effortless sharing and movement of functionality between application components. We give an implementation of our language and demonstrate its applicability by using it to implement several important components of distributed web applications, including RDBMS integration, load balancing, and fine-grained sandboxing of untrusted third party code.

The rising popularity of cloud computing and heterogeneous computer architectures is putting a strain on conventional programming models, which commonly assume that one application executes on one machine, or at best on one out of several identical machines. With our language, we take the first step towards a programming model better suited for a computationally multicultural future.

1 Introduction

1.1 Background

Conventional programming models often assume that one program corresponds to one application. With the rise of cloud computing and increasingly heterogeneous computing platforms, this assumption is becoming increasingly invalid. Modern applications, and web applications in particular, are more often than not distributed affairs, involving multiple nodes which often have very different programming models.

One example of this would be the so-called *smart home*, where refrigerators, light bulbs, toasters and a myriad of other household objects are all connected to a central hub, which allows the whole home to be controlled remotely, or even automatically in response to data gathered in real-time from the various objects. Not only does a smart home application need to reliably integrate any number of distributed components, but those components generally have very different capabilities, and may even span several different networks. The central hub might be connected to the Internet for remote control, some sensors may actually be a smarter front-end for an array of "dumb" sensors, and so on.

However, languages and programming models are slow to catch up. Distributed applications are conventionally written, not as a single program, but as a set of independent programs communicating over a set of home-grown, more or less ad hoc protocols. While this model allows developers some flexibility in choosing the appropriate language for each component individually, it has several serious shortcomings:

- Code cannot be easily shared between components. Even when two components are written in the same language, sharing functionality between them involves adapting the build system or taking care to break shared functionality into libraries, even for minor, otherwise unrelated code fragments.
- Applications are not type-safe. Changes in one component without matching changes to another, which in a non-distributed application would have triggered a type error at build time, may easily slip into production as protocol incompatibilities between components. This can be particularly problematic – dangerous, even – for applications which make use of actuators to affect the physical world, as in the case of smart homes.
- Distribution requires significant overhead: network communication needs to be implemented, protocols designed, and networks configured. Even worse, this fragile machinery needs to be properly maintained, adding an extra cost to refactoring and bug fixing, as well as to feature development.

This forces developers into *early design decisions*. Once a piece of functionality is implemented in a particular component, it cannot be easily moved to another. However, deciding how and where any particular functionality should be run up front may not be optimal. Design decisions are best made with as much information available as possible; information which often becomes available only as a project makes gradual progress.

Heterogeneity and Domain-Specific Languages In functional programming circles, *embedded domain-specific languages*, or EDSLs, are a popular design pattern for dealing with heterogeneity with regards to an application's problem domain, letting domain experts, who may not be experts in the EDSL's host language, productively express problems and their solutions in code, without having to take a detour via a host language expert. EDSLs are also being used to let developers program systems with heterogeneous components, such as GPUs or special-purpose many-core processors without having to leave the host language [Karácsony and Claessen, 2016, Mainland and Morrisett, 2010].

While helpful in managing local heterogeneity, EDSLs usually do not do much to help with *distributed* heterogeneity. Remotely executing code written in an EDSL often boils down to writing an ad hoc, boilerplate server which accepts commands from a client, translates them into an appropriate program in the EDSL, executes the resulting program, and sends the result back to the client. This mode of development suffers, just like the more general case discussed previously, from excessive development overhead and forced early design decisions.

Web-Specific Trust Challenges While distributed applications in general often assume a model of mutual trust, in which all nodes making up the application are presumed to faithfully represent the intention of the developer, web applications are fundamentally incompatible with this assumption. By nature of executing locally on user-controlled machines, any application node representing the client part of a web application cannot be trusted not to have been tampered with. As a result, any distributed web application must treat any client nodes as adversaries.

Worse, unlike other software, web applications commonly load third-party dependencies from online sources during run-time. While developers may not always audit third party libraries as thoroughly as they perhaps ought to, with a conventional application the user can at least be reasonably sure that the code they are running is the code the developer did ship. When libraries are loaded from third-party sources at run-time, this assumption no longer holds. The application's attack surface expands to include every machine and network from which libraries are loaded. Even worse, a library developer may turn out to be less honest than initially assumed, surreptitiously replacing a previously audited library with a malicious one when least expected. For this reason, not only do any server nodes need to treat the client as an adversary, but the *client* also needs to treat *parts of itself* as an adversary!

1.2 A Language for Distributed Web Applications

This paper presents an EDSL for implementing distributed applications with heterogeneous components, with a particular focus on web applications. Our language is embedded in Haskell and uses its expressive type system to separate nodes based on their roles, capabilities, and localities. To produce binaries for an application's nodes, *the same source code* is compiled multiple times with different compilers or settings: once for browser-targeting nodes, and once for each distinct server-side binary desired. This lets an application span nodes running on any architecture that has a GHC-compatible Haskell compiler. Remote calls between nodes are synchronous and type-safe. By default, any node may call any other node, but a node may optionally declare that it may only be called by nodes that are instances of some type class. Calls may not be made to the main client node as there may be an arbitrary number of clients running – one for each user's web browser – with no way to distinguish them from each other. Programs written in third party EDSLs can be attached directly to the network by providing two simple type class instances, and integrate seamlessly with our language even in cases where there are significant differences in the type universes of the EDSL and the host language.

The Structure of this Paper In Sect. 2, we describe our language in further detail and implement several common building blocks of web applications, to give an intuition of its use and to demonstrate its flexibility. In Sect. 3, we describe the implementation of our language. In Sect. 4, we discuss in further detail the limitations, design decisions and tradeoffs of our language, as well as its relation to the previous state of the art. Finally, in Sect. 5, we note our conclusions, and discuss possible directions for future research into the topic.

Our Contribution The contribution of this paper is threefold.

- We present a domain-specific language for developing distributed web applications. Our language is embedded in Haskell and improves upon the state of the art by enabling type-safe communication between any number of networked components with heterogeneous programming models, executing on any number of different platforms including web browsers.
- We show how our language provides a common framework for integrating domain-specific languages distributed over a network, and enables late reconfiguration of application structure and topology.
- We demonstrate the viability of our language by using it to implement several common building blocks of distributed web applications, including RDBMS integration, load balancing, and sandboxed execution of untrusted third party code.

2 The Language

The discussion in Sect. 1 of the limitations of conventional languages and programming models makes it clear what we don't want to see in a language for distributed web applications. But then, what properties *do* we want such a language to have?

- We want our language to support constructing distributed applications as a single program, with boilerplate-free, type-checked communication between nodes. Nodes should be able to run on different platforms, including the user's web browser and some native server-side environment.
- Our language should also support sharing and moving code between nodes to the greatest extent possible, as well as reconfiguring the "shape" of the application with minimal fuss; adding or removing nodes, and re-shaping the network over which the nodes communicate.
- The language should enforce clear boundaries between nodes. As discussed in Sect. 1, web applications suffer from an endemic lack of trust, and any language targeting this domain should strive to make the flow of control and data between components clear, to minimise the risk of information leakage.
- Finally, the language should integrate cleanly with other domain-specific languages embedded in the host language, pre-existing as well as custom-made.

In this section, we give a brief overview of the interface and programming model of our language, and show how it fulfils said design goals.

2.1 Basic Programming Model

Our programming model is client-centric: programs in our language are written in a monad *Client*, loosely analogous to the *IO* monad inhabited by "normal" Haskell programs. Client code is compiled to JavaScript for execution in the user's web browser, constituting the main user interface of the application.

Distributed applications are constructed from a set of *nodes*, each node is made up of a data type $n : * \rightarrow *$, which is the type of computations performed on said node, and an instance of the *Node* type class, describing how the node integrates with the larger application. A computation of type $n\ a$ always executes on the node n. However, pure code is shared among all nodes, and polymorphic code can be called from any node which meets its type class constraints. Initially, *Client* is the only defined node. By default, any node which also implements a *MonadClient* type class may call any other node. However, when defining a node, the implementor may choose to put additional restrictions on which other nodes may make calls to it.

Nodes expose *entry points* as *static pointers* [Epstein et al., 2011] on the program's top level. The static pointers extension, supported by GHC since

```haskell
type Endpoint :: *
type Import   :: * → *
type RemotePtr f = StaticPtr (Import f)
type Client :: * → *
type Server :: * → *

class MonadIO m ⇒ MonadClient (m :: * → *)

class Node (n :: * → *) where
  type Allowed n (c :: * → *) :: Constraint
  type Env n :: *
  endpoint :: Proxy n → Endpoint
  init :: Proxy n → CIO (Env n)

class Node n ⇒ Mapping (n :: * → *) f where
  type Hask n f
  invoke :: Env n → n f → CIO (Hask n f)

remoteNode :: String → Int → Endpoint
localNode  :: Proxy a → Endpoint
remote     :: Export f ⇒ f → Import n f
dispatch   :: Dispatch f f' ⇒ RemotePtr f → f'
dispatchTo :: Dispatch f f' ⇒ Endpoint → RemotePtr f → f'
start      :: Node n ⇒ Proxy n → CIO ()
runApp     :: [CIO ()] → Client () → IO ()
```

Figure 5.1: The user-facing API of our language

version 7.10, introduces the *static* keyword to assign names to values that are known at compile time. Names assigned in this way are stable across the network, allowing different binaries produced from the same source to refer to each others' values.

The remainder of this section introduces our language by showing how a series of examples can be implemented using our language, the API of which is given in Fig. 5.1. Note the use of the *CIO* monad, where one would normally expect *IO*. As the Haste compiler, used by our implementation, does not support concurrency in the *IO* monad, our implementation uses *CIO* as an explicitly concurrent drop-in replacement. The *Dispatch* type class denotes any pair of nodes *f* and *f′* such that *f′* is able to make a remote call to *f*. This type class is covered in greater detail in Sect. 3.3.

To give an intuition of the basics of our language, Fig. 5.2 shows a "hello world" application with two nodes, the web-based client, and a single node running on a server, where the client uses a remote pointer to print a message on the server. Lines 1 and 2 insert the *Server* node into our application, specifying that it can be reached on the host `example.com`, on

```
1  instance Node Server where
2    endpoint _ = remoteNode "example.com" 24601
3
4  greet :: RemotePtr (String → Server ())
5  greet = static (remote $ liftIO . putStrLn)
6
7  main = runApp [start (Proxy :: Proxy Server)] $ do
8    dispatch greet "Hello, server!"
```

Figure 5.2: Hello, server!

port 24601. This information is referred to as the node's *endpoint*. Note the use of the *static* keyword on line 5, which together with the call to *remote* introduces a remote pointer, and the *dispatch* function on line 8, which is used to remotely invoke the greeting function. The invocation of *start*, passed to the *runApp* function on line 7, indicates that any non-client binary of this application can handle requests to the *Server* node. The second argument to *runApp* is the entry point of the application's *Client* computation.

In a conventional web application, even this simple program would have required significant legwork: separate programs need to be written for client and server, a network transport protocol needs to be chosen, a data serialisation format decided upon and implemented, and client-server communication code needs to be implemented. A lengthy and error-prone task, which is not made easier by the lack of type safety between the client and the server.

The discerning reader may find the syntax of remote imports – the *static* keyword and the way it combines with the *remote* function – to be slightly clunky. Regrettably, this syntax is forced upon us by the use of static pointers. However, in addition to allowing us to make use of static pointers, this syntax has the distinct advantage of making it crystal clear where the boundaries between nodes are. These sharp boundaries are helpful in avoiding accidental information leaks between nodes.

2.2 Compiling and Executing Programs

As our language is embedded into Haskell, compiling and executing programs follows the normal Haskell compilation workflows. There is, however, a twist: programs written in our language need to be compiled *several times*, possibly with different compilers, depending on the number of nodes they contain as well as their respective architectures.

The program in Fig. 5.2 is relatively simple, and consists of two components: the client which sends the greeting, and the server which receives it. This application needs to be built twice: once with a Haskell compiler which produces a JavaScript program, and once with a compiler which produces a native binary. Once built, the application is started by executing the native binary on `example.com`, and by loading the JavaScript program in a web browser.

For less trivial applications, spanning more than a single server node, the application may need to be built several times, up to once for each type of node in the application. In this case, each build should change the type of the *start* invocation to match the type of the node that is currently being built. However, each native binary may run any number of nodes, by adding an appropriately typed invocation of *start* for each node it intends to run, and a single binary may be run on more than one physical machine.

2.3 A "Real" Application: State and Third-Party EDSLs

We now move on to a more complex application: a photo browsing site, where users may search for photos by description and discuss the photos in a real-time chat room. This example demonstrates the use of server-side state, as well as how to add a third-party EDSLs as just another node in the application. To monetise our hard work, we also include advertisements in our application, demonstrating how code from a third party may be safely included in an application using our language.

Fig. 5.3 shows the architecture of our example application. Note that while this application only consists of a client talking to multiple servers, our language allows servers to call each other as well, as discussed in Sect. 2.1.

Integrating with Exotically Typed Third-Party EDSLs To allow users of our application to search for images, we use a relational database to store pairs of image identifiers and their corresponding full-text descriptions. For this purpose, we use our language to incorporate the arrow-based *Opaleye* [Ellis, 2014] SQL EDSL as a node in our application. The use of Opaleye in this example is entirely arbitrary. As previously discussed in Sect. 2.1, any EDSL, third-party or in-house, may be used as a node in our language.

Opaleye is what we call an *exotically typed* language. That is, the return type of a computation *within* the Opaleye EDSL, is not the same as the type of the value produced by actually *running* the query, reflecting an impedance mismatch between the Opaleye and Haskell type universes. This impedance mismatch is not specific to Opaleye or even to relational database EDSLs in general, but is common among deeply embedded EDSLs:

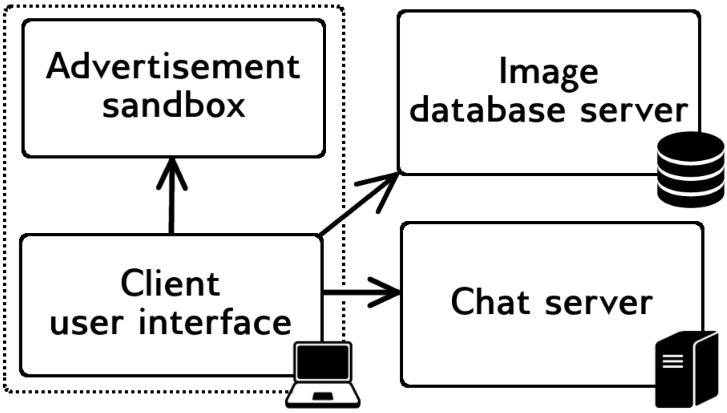

Figure 5.3: Architecture of our example application

the *Nikola* [Mainland and Morrisett, 2010] EDSL for GPU programming, and the *Aplite* [Ekblad, 2016] and *Sunroof* [Bracker and Gill, 2014] EDSLs for web development also share this trait, just to name a few.

Our language lets us seamlessly integrate such EDSLs using the *Mapping* type class, first seen in Fig. 5.1. This type class allows us to define on a node by node basis, for each type *t* in that node's type universe, a corresponding type *Hask t* in the type universe of the host Haskell program. To enable translating values returned from such nodes, *Mapping* also lets us define a function *invoke*, which describes how computations on this particular node should be executed.

Opaleye requires us to first define the structure of any tables we want to use. We define a table consisting of two fields: an image's *id* and its *description*.

```
photoTable :: Table (Column PGInt4, Column PGText)
                    (Column PGInt4, Column PGText)
photoTable = Table "photos" (p2 (required "id", required "description"))
```

Then, we can use Opaleye's *Query* arrow to create a function which lets us perform a remote full-text search in our image database.

```
findImage :: RemotePtr (String → Query (Column PGInt4))
findImage = static (remote $ \key → proc() → do
    (id, desc) ← queryTable photoTable -< ()
    restrict -< desc ‘like‘ pgString key
    returnA -< id)
```

Note the return type of this remote function. Our EDSL has no built-in idea of how to deal with values of type *Column PGInt4*, but we still want to be able to import this function.

For our image search database, we need to map *Column PGInt4* to *[Int]*, like in Opaleye proper, and to execute queries over the appropriate database file. After adding this *Mapping* instance, all we need to create a fully functional database node is the customary *Node* instance.

```
instance Mapping Query (Column PGInt4) where
  type Hask Query (Column PGInt4) = [Int]
  invoke _ q = liftIO $ do
    withConnection "images.sqlite" $ \c →
      runQuery c q
instance Node Query where
  endpoint _ = remoteNode "db.example.com" 24601
```

Stateful Servers To support type-safe handling of server-side state, we introduce two new members of the *Node* type class: an associated type *Env*, and a method *init*. *Env* specifies the *environment type* of a node, and defaults to () – denoting an empty environment – unless explicitly given. The method *init* is executed when a node is started, and allows programmers to perform any initialisation, before creating and returning the node's environment. From here on, the environment is *immutable*. An immutable environment may contain any kind of mutable synchronisation primitive, such as *IORef*s or *MVar*s. By only providing a well-defined method to access any such constructs, we let the developer choose which kind of synchronisation fits their application best, instead of forcing some arbitrary storage upon them.

In our application, we need two items in our environment. A mutable reference to a list of strings, representing all lines sent to chat so far, and a list of pairs of user names and password hashes, as we only want to allow users with proper credentials to send messages. In our *init* function, we initialise this environment to contain no chat lines as of yet, and to have a single user, *john*, with a spectacularly bad password. As usual, we also set up an endpoint at which the chat server can be reached.

```
type State = (IORef [String], [(String, Hash)])
type ChatSrv = ReaderT State Server

instance Node ChatSrv where
  type Env ChatSrv = State
  init _ = liftIO $ do
    msgs ← newIORef []
    return (msgs, [("john", hash "password1")])
  endpoint _ = remoteNode "chat.example.com" 24601
```

Now, we are ready to define the two operations supported by the chat

```
postMessage :: RemotePtr (String → String → String → ChatSrv ())
postMessage = static (remote $ \user pass msg → do
    (ref, creds) ← ask
    case lookup user creds of
      Just db_hash | db_hash == hash pass → do
        liftIO . atomicModifyIORef' ref $ \ms →
          ((user ++ ": " ++ msg):ms, ())
      _ → throw (AuthError "bad login"))

checkMessages :: RemotePtr (ChatSrv [String])
checkMessages = static (remote $ ask >>= liftIO . readIORef . fst)
```

Figure 5.4: Implementation of the chat server

server: *postMessage* and *checkMessages*. *postMessage* takes a user name, a password and a message. It posts the given message to the chat, prefixed with the sender's name, if the sender has the appropriate credentials. *checkMessages* simply fetches all messages sent to the chat server so far, assuming the user has the requisite credentials. Fig. 5.4 gives the implementation of the chat server.

It should be noted that our implementation of *checkMessages* is not optimal: instead of polling for messages to download, a proper implementation of this function would block waiting for new messages to arrive. We have opted to use a "dumb" implementation here, to keep our example application as simple as possible.

Tying it Together: the Client As we use the Haste compiler to compile our client nodes, it is natural that we also use Haste's GUI library for our user interface. In keeping with the theme of using domain-specific languages wherever possible, the *structure* of the user interface is defined in an external HTML file, with only its *behaviour* given here. Fig. 5.5 gives the implementation of the client.

The code in Fig. 5.5 starts by fetching the widgets associated with the HTML identifiers *search*, *img*, etc. On lines 9 to 17, an event handler is set on the *search* input, to search the image database for an image matching the description given in the input field. On lines 19 to 24, an event handler is set on the text box where chat messages are typed, to send a message when the user hits return while the input if focused. Finally, we set up a timer to poll the server for new messages once every second. As previously noted, the polling model is suboptimal and easily replaced by a more sophisticated solution, but was chosen for our example in the interest of brevity.

```
main = runApp
    [ start (Proxy :: Proxy ChatSrv)
    , start (Proxy :: Proxy Query)
    ] $ do
  withElems
    [ "search", "img", "box", "chat", "user", "pass"
    ] $ \[search, img, box, chat, user, pass] → do
      search ‘onEvent‘ KeyUp $ \key → do
        when (key == keyReturn) $ do
          term ← getProp search "value"
          imgs ← dispatch findImage term
          case imgs of
            (i:is) → set img ["src" =: (show i++".jpg")]
            _      → alert "No matches for that term."

      chat ‘onEvent‘ KeyUp $ \key → do
        when (key == keyReturn) $ do
          [u, p, m] ← mapM (flip getProp "value") [user, pass, chat]
          setProp chat "value" ""
          dispatch postMessage u p m

    setTimer (Repeat 1000) $ do
      msgs ← dispatch checkMessages cycle
      setProp box "value" (unlines msgs)
```

Figure 5.5: Combined image search and chat client

Late Design Decisions: Moving Code Between Nodes One advantage of our programming model is the ease of moving code between similar nodes. Moving code between two completely different nodes – say, the database and the chat server – would be hard due to their differing programming models, but moving code between semantically compatible nodes is as easy as refactoring any non-distributed application. For instance, to protect the confidentiality of the user's password from the possibly untrusted server operator in the chat server example, one would simply need to move the call to *hash* from the *postMessage* remote function to the client. Similarly, user authentication could be offloaded to a dedicated server simply by adding a new function of the appropriate type, with the appropriate *Node* instance. In this way, developers can easily experiment with different architectures and implementations.

2.4 Sandboxing Third-Party Code

Recall from the discussion in Sect. 1, that web applications must often assume *parts of themselves* to be potentially malicious. To combat this problem, web browsers provide a mechanism called *sandboxed iframes*: a compartment in which sections of code can be executed, without being able to inspect or influence the rest of the application. Unfortunately, this construct is relatively clunky. Code executing in a sandboxed iframe can only communicate with the larger application through explicit message passing. As the JavaScript execution model is explicitly non-concurrent, both the sandboxed code and the application at large effectively need to be written in continuation-passing style. Additionally, glue code needs to be written to allow third party libraries – which usually follow a standard imperative programming model and definitely don't expect to deal with message passing – to execute in a sandbox, and to allow the larger application to create and communicate with the sandbox as well. For this reason, sandboxing is relatively unusual in conventional web applications, and is often limited to encapsulating relatively large, monolithic components.

Our language improves upon this situation by allowing a sandboxed iframe to be integrated into an application as just another node. Similar to how programs are split at compile-time, to produce one web client and any number of native servers, the *web client itself* is "split" at *run-time* to produce one "true" client, as well as multiple "server" nodes, each running in its own sandboxed iframe. The web browser's built-in sandbox implementation ensures that each sandbox is unable to communicate with the rest of the application except by responding to requests from the client. This lets us isolate not only external third-party code, but *any* piece of code, which enables developers to move more code from the trusted code base of the main application into the untrusted and locked-down environment provided by the sandbox. Like other nodes, sandboxes are identified by their type. Once created, a sandbox persists for the duration of the application, allowing sandboxed code to keep its state between invocations.

Sandboxed nodes are *local* to the client, as importing JavaScript client code onto a random server does not make much sense. For this reason, it should only be callable by the client. This is easily accomplished by setting the *Allowed* associated type class constraint of the *Node* class to only match the specific node *Client*.

In Fig. 5.6, we augment the chat application from section 2.3 with an advertisement, which is updated every 10 minutes by code from a third-party ad service. Since the ad rotation code is executed in a unique sandbox, parameterised over the *AdRotation* type, it can't interfere with the rest of the application or code running in other sandboxes, ensuring the integrity of the user's passwords, messages and image browsing habits.

```
data AdRotation
type MySbx = Sandbox AdRotation

instance Node MySbx where
  endpoint = localNode
  type Allowed MySbx a = a ~ Client
  init = dependOn ["http://example.com/ads.js"]

randomAd :: RemotePtr (MySbx URL)
randomAd = static (remote $ liftIO . ffi "getAdvert")

main = runApp [start (Proxy :: Proxy MySbx), ...]
  ...
  setTimer (Repeat (10*60*1000)) $ do
    adURL ← dispatch randomAd
    withElem "ad_banner" $ \banner_img → do
      setProp banner_img "src" adURL
```

Figure 5.6: Adding sandboxed ads to the chat client

2.5 Load Balancing and Dynamic Dispatch

While having a single node type correspond to a single server has several merits, such as centralising endpoint configuration and reducing communication boilerplate, there are several use cases where this model is a poor fit.

Often, a single physical server per logical node is not enough to meet the computational needs of the application, requiring tasks to be *load balanced* across multiple servers. This requires the ability to deploy several identical nodes of the same type, randomly choosing which node to call at run-time.

For interactive or bandwidth-intensive applications, it is often advisable to station servers at strategic locations throughout the world, and serve content from the server closest to each user, as is commonly done by content distribution networks. Like load balancing, this requires several identical nodes to be deployed, but now the call routing is no longer random but informed by external factors.

Certain services, such as in-home media streaming solutions, run on a user's local network, but registers with a central server from which the user can find the service without having to know its exact address on their home network. Implementing such an application in our language would involve fetching the endpoint for a node at run-time, via request to *another* node.

To accommodate these use cases, we provide a companion to the *dispatch* function, which takes the endpoint as an extra argument in addition to

```
instance Node Server where
  endpoint _ = remoteNode "1.example.com" 24601

balance :: MonadIO m ⇒ [Endpoint] → m Endpoint
balance endpoints = do
  i ← liftIO $ randomRIO (0, length endpoints-1)
  return (endpoints !! i)

work :: RemotePtr (String → Server String)
work = static (remote performHeavyWork)

main = runApp [start (Proxy :: Proxy Server)] $ do
  ep ← balance
    [ remoteNode (show n ++ ".example.com") 24601
    | n ← [1..5]
    ]
  result ← dispatchTo ep work "[a massive data set]"
  presentResult result
```

Figure 5.7: Load balancing using dynamic dispatch

the the function to be dispatched: *dispatchTo*. When using this function, the provided endpoint *overrides* the endpoint defined by the node's *Node* instance. Fig. 5.7 shows an example implementation of load balancing using the *dispatchTo* construct. A "heavy" task is distributed randomly across five different servers, located at *n*`.example.com`, all running an instance each of the *Server* node.

3 Implementation

This section describes the implementation of our language, abstracting over some implementation details: the minutiae of network communication and event handling. The interested reader may refer to our full implementation, freely available from our website[1].

3.1 Prerequisites and Assumptions

Our implementation targets the de facto standard GHC Haskell compiler, to produce server-side binaries, and the Haste [Ekblad, 2015] compiler to produce browser-based client code. Haste is a GHC-based, JavaScript-targeting Haskell compiler, which mainly differs from GHC Haskell in that concurrency is simulated within the previously mentioned *CIO* monad,

[1]https://haste-lang.org

```
class Serialize a where
  toJSON   :: a → JSON
  fromJSON :: JSON → Maybe a
class MonadIO m ⇒ MonadClient m where
  call :: Endpoint → StaticKey → [JSON] → m JSON
```

Figure 5.8: Assumed functions

as it targets non-concurrent JavaScript implementations. Our language is implementable on any recent GHC-based Haskell compiler, but relies crucially on several language extensions.

- Our language relies on *static pointers* [Epstein et al., 2011] to allow users to export remote functions.
- *Scoped type variables* and *type families*, both open and closed [Chakravarty et al., 2005, Eisenberg et al., 2014], are used extensively in our implementation to configure nodes and to implement remote function dispatch.
- The *CPP* extension, enabling the use of the standard C preprocessor, is used to implement program splitting into client and server programs.
- *Flexible instances*, *overlapping instances*, *flexible contexts*, *constraint kinds* and *multi-parameter type classes* are all used to ease Haskell's normal constraints on type class instances and constraints.
- *Default signatures* are used for certain methods and types of the *Node* type class, to reduce the amount of boilerplate necessary for common use cases.

Our implementation uses JSON as its data serialisation format. To abstract over the details of data serialisation, we assume the existence of a type class *Serialize*, which allows the conversion of values to and from a JSON-encoded string.

We also assume the existence of a type class *MonadClient*, for each node specifying how a packet representing a remote call is sent from one networked machine to another, to abstract over the details of network communication. Instances of this type class are also responsible for converting network errors and remote exceptions sent in reply to requests into errors appropriate for the instance. A listing of assumed functions is given in Fig. 5.8.

3.2 Exporting Remote Functions

In order to make a remote function available to clients, it must first be converted to a *remote import*. A remote import consists of a static pointer to a function from an environment for the node on which the import is intended to execute and a list of serialised arguments, to a serialised return value in the *CIO* monad previously discussed in Sect. 2:

```
newtype Import f = Import (Env (Aff f) → [JSON] → CIO JSON)
```

In our implementation, we need to refer to the node on which a function is intended to execute, the function's *affinity*, and the value resulting from fully applying the function and executing the resulting computation on its affinity node, its *result*. We define two closed type families to extract this information.

```
type family Aff f :: * → * where
  Aff (a → b) = Aff b
  Aff (n a)    = n
type family Res f :: * where
  Res (a → b) = Res b
  Res (n a)    = a
```

We then use these two type families to define a type class *Remote* with a single method *toRemote*, to transform any exportable function into another function from a list of serialised arguments to a *CIO* computation returning a *serialisable* value. The conversion is completed by the *remote* function, which constitutes the user interface for exporting functions as discussed in Sect. 2. An exportable function is any function where all of its arguments are serialisable, and where the *mapping* of its result – the host Haskell type corresponding to the function's possibly domain-specific result type – is also serialisable. The *Export* constraint captures this concept more precisely:

```
type Export f = (Remote (Aff f) f, Serialize (Hask (Aff f) (Res f)))
```

Fig. 5.9 gives the implementation of the *Remote* type class and *remote* function. The *toRemote* function recurses over the argument types of the function to be exported, in each step applying it to the next in a list of serialised values provided by the caller. Once the exported function has no more arguments, we have reached our base case. At this point, we simply run the fully applied function using the *invoke* function. As discussed in Sect. 2, *invoke* can be seen of as a node's "runner function" – a function that executes an EDSL computation, returning some result back to the host language – and is given by each node's instance of the *Mapping* type class. This basic method used to transform one polymorphic function into another is well known, and is used by, among others Ekblad [2016], to transform

```
class (Aff f ~ n) ⇒ Remote n f where
  toRemote :: f → Env n → [JSON] → CIO (Hask n (Res f))

instance (Serialize a, Remote n b) ⇒ Remote n (a → b) where
  toRemote f env (x:xs) | Just x' ← fromJSON x = toRemote env (f x') xs
  toRemote _ _ _ = throw CommunicationError

instance (Aff (n a) ~ n, Res (n a) ~ a, Mapping n a) ⇒ Remote n (n a) where
  toRemote f env _ = invoke env f

remote :: ∀f. Export f ⇒ f → Import f
remote f = Import $ \env xs →
  let c = toRemote f env xs :: Hask (Aff f) (Res f))
  return (toJSON c)
```

Figure 5.9: Transforming functions into remote imports

function written in the Aplite EDSL into functions callable from the Haskell host – a use case quite similar to ours.

3.3 Dispatching Remote Calls

The remote import conversion is a server-side process, which converts eligible functions into remote imports *before* exporting them as static pointers. Similarly, there is some conversion needed on the side of the caller before those static pointers can be used to make remote calls. This conversion is in some sense the inverse of the remote import conversion.

Again, we recurse over the arguments of the remote function, this time using the *Remotable* type class, with its single *fromRemote* method. This time, however, we do not gradually apply the function, but instead we gradually *build up* the function that gathers its arguments, in serialised form, in a list. Once we reach the base case – again, when the remote function has no arguments left to process – we dispatch the remote import with the argument list we built up during the recursion, and wait for the server's reply to arrive. Fig. 5.10 gives the complete implementation of remote function dispatch.

The user-facing interface to this machinery is the *dispatch* function. While its implementation is simple, only consisting of a single call to *fromRemote*, its type is more interesting; more specifically, the *Dispatch* constraint. Some of the sub-constraints of *Dispatch* are expected: that remote functions must be actually remotable, and that their return types must have a mapping to equivalent host Haskell types – recall the discussion on exotically typed nodes in Sect. 2. However, the constraint that the type *HaskF (Aff f') f* must

```
class Remotable a where
  fromRemote :: Endpoint → StaticKey → [JSON] → a

instance (Serialize a, Remotable b) ⇒ Remotable (a → b) where
  fromRemote e k xs x = fromRemote e k (toJSON x : xs)

instance (MonadClient from, Serialize a) ⇒ Remotable (from a) where
  fromRemote e k xs = do
    res ← fromJSON <$> call e k (reverse xs)
    case res of
      Just r → return r
      _      → throw NetworkError

type family HaskF from a where
  HaskF from (a → b) = (a → HaskF from b)
  HaskF from (m a)    = from (Hask m a)

type Dispatch f f' =
  ( Node (Aff f)
  , Allowed (Aff f) (Aff f')
  , HaskF (Aff f') f ~ f'
  , Remotable f')

dispatch :: ∀f f'. Dispatch f f' ⇒ RemotePtr f → f'
dispatch f = fromRemote ep (staticKey f) []
  where ep = endpoint (Proxy :: Proxy (Aff dom))

dispatchTo :: Dispatch f f' ⇒ Endpoint → RemotePtr f → f'
dispatchTo ep f = fromRemote ep (staticKey f) []
```

Figure 5.10: Dispatching remote functions

be equivalent to f is, perhaps, slightly opaque.

The *HaskF* type family extends the concept of host Haskell equivalent types to function types: if *Hask n a* denotes the equivalent host Haskell type of domain-specific type a for some node n, then *HaskF n f* denotes the equivalent host Haskell type of any function $f : a \to \ldots \to n\ b$ for some domain-specific type b and some node n. Note that the *argument types* of f are here assumed *not* to be domain-specific. The reason for this is that most EDSLs are easily able to accommodate conversion from host values to domain-specific values within the language itself, whereas conversion of return values are entirely dependent on an EDSL's runner function.

3.4 Servers and Program Splitting

The server-side implementation of remote calls is comparatively simple. For each node, its *init* method is invoked once to perform any setup necessary and to obtain its environment. Then, a thread is started to listen for incoming connections on the port specified by the node's endpoint. When a remote call arrives at a node, the node attempts to de-serialise it. If it could not be de-serialised, or if the static pointer denoting the call's entry point is invalid or does not match its accompanying arguments, an error is reported back to the caller where it is raised as an exception. This may happen if, for instance, there is a version mismatch between the deployed client and server, or if the client program has been modified by a malicious actor. Otherwise, the static pointer to the remote import is de-referenced to obtain the computation to be performed. The computation so obtained is invoked with the node's environment and the argument list received with the remote call. If no error is raised during execution of said computation, the result is sent back to the caller. If an error *is* raised during execution, the error is instead reported back to the caller.

More interesting, then, is the implementation of how nodes are *started*, as this is where programs are split to determine where each node should run: on a native server, in a sandbox, on the client, or not at all? This splitting happens in two steps. First, at compile-time, native nodes are separated from nodes intended to run in a browser. This is accomplished by subtle differences in the implementations of the *start* and *runApp* function, depending on which compiler was used to build our language. If the program was built using the web-targeting compiler, the *Client* computation given to *runApp* is executed, but the *start* computations for any remote nodes are not. Conversely, when built using a native-targeting compiler, remote nodes *are* started, while the main client computation isn't.

The second step happens at run-time, and separates the client computation from any other client-side nodes – that is, the nodes running in sandboxes. The principle is the same as in the web/native situation, but instead of splitting programs based on their compiler environment, programs are split based on their *browser* environment. The JavaScript program produced by our web-targeting compiler is executed both in the client browser context and in any sandboxes.

If the program detects that it is not running in a sandbox, it proceeds to create an iframe sandbox for each sandbox node. The sandboxes are then set up to execute *the same* program, after which the program – still outside the sandboxes – proceeds to execute the *Client* computation passed to *runApp*. If the program instead detects that it is indeed inside a sandbox, it sets up an event handler to wait for requests from the main client node and then goes dormant until a request arrives.

4 Discussion and Related Work

4.1 Our Language as a Collection of Remote Monads

It is worth noting that nodes in our language are almost, but not quite, instances of the *remote monad* design pattern identified by Gill et al. [2015]. We say not quite, because in the remote monad design pattern computations are composed on the client and sent off to a server for execution. In light of the web-specific issues highlighted in Sect. 1, this is not desirable in a language targeting web applications. Consider the following function.

```
safelyLaunchMissiles :: Credentials → Server ()
safelyLaunchMissiles cred = do
  can_launch_missiles ← isPresident cred
  when can_launch_missiles launchMissiles
```

If this computation is exposed to the client as an atomic unit, there is no problem: if anyone but the president attempts to fire missiles, the permission check fails and the missiles stay where they are. However, if this computation was assembled on the client and only then dispatched to the server, a malicious user might simply *remove* the permission check completely! For this reason, we do not allow clients to arbitrarily compose computations before executing them remotely, but instead only allow computations exported via static pointers to be remotely executed.

4.2 Relation to Microservice-Based Architectures

Microservices is another, increasingly popular, methodology for writing distributed applications, which takes a radically different approach from our language. Applications are written explicitly as a collection of small components communicating over some network protocol or message bus. Even components which may traditionally be seen as monolithic programs are usually split up, to achieve the highest possible granularity. While this approach achieves excellent decoupling, it offers very few, if any, safety guarantees, as each component is considered to be a truly independent program. It does, however, offer good reliability properties, as components are split into separate processes, possibly running on different machines.

Our language sacrifices a degree of decoupling in exchange for type safety. However, this does not mean that each application must be deployed as a monolith. As long as the type signatures of the entry points exported by a node in our language do not change and all remote imports are declared on the top level, said node – and any nodes that call it – may be independently updated and deployed. Leveraging this property and compiling each node into a separate binary, as described in Sect. 2.1, an application written in our language can achieve the same degree of reliability as an equivalent microservices-based application, but not the same level of decoupling.

4.3 Simplicity and Flexibility

In the design of our language, we have attempted to strike a balance between flexibility and simplicity. Often this tradeoff has taken the shape of a more flexible, but more complex, behaviour for the general case, with sensible defaults for common special cases. The *Node* type class, first introduced in Sect. 2, and the accompanying *Mapping* class, is perhaps the prime example of this design philosophy. While the combination admits considerable flexibility – crucial to our support for exotically typed EDSLs – type class defaults reduce the amount of complexity required to implement simpler applications to a significant degree.

When a more complex, possibly desirable behaviour can be implemented on top of a simpler one with relatively little hassle, we have opted for the less complex option. *Push notifications* are an instance of this design choice. Recall that our language is client-centric, and does not allow calls to be made to the *Client* node. At first glance, this would seem to make push notifications impossible. However, even though this mode of operation is not supported intrinsically by the programming model, it is easily implemented on top of our language using concurrency and *long polling* – a technique where a synchronous remote call blocks indefinitely, until the server is ready to provide a reply.

4.4 Related Work

Multi-tiered programming models for web applications have been a popular research topic of late. The problems with JavaScript as an implementation language for complex applications are well known, as are the advantages of using a coherent client-server framework to eliminate ad hoc network protocols and error-prone boilerplate code. Most existing attacks on this problem assume a relatively simple problem description: a web application consists of a client communicating with a single server, both of which share the same programming model.

However, with modern web applications, this assumption often does not hold. Not only can the traditional "server" of a multi-tiered web application often be broken down into domain-specific "sub-servers" – a conventional server which stores large files and a database server which stores metadata is a very common case – but clients themselves in reality often talk to a multitude of servers apart from the canonical one. Functionality to, say, geographically locate users or look up the postal code of every town on earth, resides on some third party server, a credit card payment processor resides on a second, and advertisements are served from a whole network of them.

This section attempts to give an overview of the field of multi-tier web application languages, comparing our language to the current state of the art.

Eliom The *Eliom* [Radanne et al., 2016] OCaml dialect provides a programming model for client-server web applications in which components are allocated to either the client or to a single server using explicit annotations. An interesting property of its programming model is that it is not only multi-tiered, but *multi-staged* as well: server-side code can manipulate client-side data as first class values, enabling a form of partial evaluation. Eliom also acknowledges the fact that the type universes of the client and the server may be different, with some types having different representations and some only existing on either the client or the server. Any type that is to be passed between client and server needs an explicit *converter*, to ensure proper representation on both sides of the rift. This concept is quite similar to the *mapping* concept used by our language to enable the use of exotically typed EDSLs.

Haste.App and AFAX *Haste.App* [Ekblad and Claessen, 2014] uses a similar approach to our language, in which clients and servers are separated based on their type. Also like our language, Haste.App is embedded in Haskell, and uses compile-time program splitting to enable the client and the server to be produced by compiling the same application with two different compilers. *AFAX* [Petricek and Syme, 2007] takes a very similar approach to Haste.App, but is embedded in F#. AFAX also requires certain minor changes to the type system of F# in order to be truly type-safe. Neither language supports more than a single server, which is assumed to have the same programming model as the client.

JavaScript Dialects The *Conductance* application server [Cuthbertson, 2014], and the *StratifiedJS* JavaScript extension upon which it is built equip the JavaScript language with an array of new features, including concurrency and a tierless programming model. This model allows the client to make remote procedure calls to the server, but also enables communication through the use of shared mutable variables. This model is less explicit about the client-server separation than the models we have seen so far, and is perhaps the most "JavaScript-like" of the lot. *Opa* [Bourgerie, 2014], a competing framework, provides a similar programming model but dispenses with the remote procedure calls, instead basing all its communication on implicit data flows enabled by shared mutable variables. Opa also includes dedicated syntax for database queries, an important special case of domain-specific web application components.

Hop.js [Serrano and Prunet, 2016] is a JavaScript dialect which compiles down to the *Hop* language, a Scheme dialect which in turn compiles down to JavaScript, for the client part, and a native binary for the server part. It supports Eliom-like multi-stage programming and inline HTML fragments – another special case of domain-specific components. Remote communication with Hop.js is centered around the concept of *services*: first-class objects representing remote machines. While the client may only invoke the services of the server from which it was served, that server may in turn invoke other servers belonging to the application.

Neither of these JavaScript dialects provide type safety or the ability to integrate multiple programming models in a single language.

Java-Style Remote Method Invocation Our language is loosely related to Java's *Remote Method Invocation* [Downing, 1998]. This programming model allows programs to be automatically split into different servers, where each node can call methods on objects from another, as long as they possess a reference to such a remote object. RMI supports any number of nodes, but the relatively coarse-grained type system of Java forces all servers into the same programming model. The boundaries between nodes are also relatively floating, making it hard to determine where one node stops and another begins. RMI also does not provide a means for nodes to restrict which nodes may make calls to them, making it impossible to implement, for instance, the sandboxing from Sect. 2.4 using RMI.

Stand-Alone Languages Additionally, there are several stand-alone domain-specific languages for multi-tier web applications; *Links* [Cooper et al., 2007], *Ur/Web* [Chlipala, 2015] and *ML5* [Vii et al., 2007] being the more prominent ones. A property common to these languages are their focus on the three conventional tiers of web applications: client, server, and database. Consequently, they include some form of support for all three tiers, including type-safe communication between them. While this approach does acknowledge the need for a domain-specific language for web programming, little heed is paid to the possible benefits of also using domain-specific languages in the web application's *business domain*. Links and Ur/Web only support a single server, while ML5 supports multiple, homogeneous servers. From a practical standpoint, stand-alone languages are at a disadvantage compared to an embedded language or a dialect of an existing one. Not only do they miss out on the opportunity to leverage existing ecosystems in general, but supporting heterogeneous servers becomes both easier and more advantageous in the presence of a large number of pre-existing, third-party EDSLs and libraries.

5 Conclusions and Future Work

We have presented an embedded, domain-specific language for building web-targeting, distributed applications out of heterogeneous components. Our language supports seamless, type-safe integration of components written in a variety of *other* embedded, domain-specific languages, in addition to components written in plain Haskell. We improve upon the state of the art by supporting an arbitrary number of heterogeneous nodes, with automatic communication access control enforced by the type system, and by allowing seamless, boilerplate-free integration of exotically typed EDSLs.

We have demonstrated the flexibility of our language by giving sample implementations of several important web application building blocks, including load-balancing and isolation of untrusted third-party code. Our language enables developers to easily share and move code between nodes, allowing design decisions regarding application topology and configuration to be pushed later into the development process. While we have not yet made any rigorous evaluation of our language, it has been used to good effect in the implementation of an educational proof assistant, deployed in a course of over 200 students. Block et al conducted an evaluation of a previous, less capable, version of our language for the use of online multiplayer board games, with generally positive results [Block et al., 2016].

Future Work Our requirement that each node of an application is able to run a full Haskell binary can be problematic in some application areas, such as the most low-powered spectrum of the Internet of Things domain. For these areas, we intend to extend our programming model by generating platform-specific C code, including scaffolding and communication code, for nodes intended to run on low-powered devices, enabling them to take part in the same distributed applications as their full Haskell counterparts.

Our language currently only supports relatively rudimentary handling of runtime errors and reliability failures: throw an exception to which the client may choose to react. Exploring how errors may be handled in smarter ways, such as the possibility of automatically repairing certain classes of errors, looks like a promising line of future work.

Acknowledgements

Many thanks to Koen Claessen, Simon Marlow and Michal Palka for their valuable insights, feedback and discussion.

6 Bibliography

B. Block, J. Gustafsson, M. Milakovic, M. Nilsen, and A. Samuelsson. Evaluating haste.app: Haskell in a web setting. effects of using a seamless, linear, client-centric programming model, 2016.

Q. Bourgerie. The opa framework. http://opalang.org/, 2014.

J. Bracker and A. Gill. Sunroof: A monadic DSL for generating JavaScript. In *Practical Aspects of Declarative Languages*, volume 8324. Springer International Publishing, 2014. ISBN 978-3-319-04131-5.

M. M. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ACM SIGPLAN Notices*, volume 40. ACM, 2005.

A. Chlipala. Ur/web: A simple model for programming the web. In *ACM SIGPLAN Notices*, volume 50. ACM, 2015.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*. Springer, 2007.

T. Cuthbertson. The conductance application server. http://conductance.io/, 2014.

T. B. Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc., 1998.

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*. ACM, 2014. ISBN 978-1-4503-2544-8.

A. Ekblad. *A Distributed Haskell for the Modern Web*. Licentiate thesis, Chalmers Institute of Technology, 2015.

A. Ekblad. High-performance client-side web applications through haskell edsls. In *Proceedings of the 9th International Symposium on Haskell*. ACM, 2016.

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*. ACM, 2014.

T. Ellis. Opaleye. https://github.com/tomjaguarpaw/haskell-opaleye, 2014.

J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*. ACM, 2011. ISBN 978-1-4503-0860-1.

A. Gill, N. Sculthorpe, J. Dawson, A. Eskilson, A. Farmer, M. Grebe, J. Rosenbluth, R. Scott, and J. Stanton. The remote monad design pattern. In *ACM SIGPLAN Notices*, volume 50. ACM, 2015.

M. Karácsony and K. Claessen. Using fusion to enable late design decisions for pipelined computations. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. ACM, 2016.

G. Mainland and G. Morrisett. Nikola: embedding compiled gpu functions in haskell. *ACM Sigplan Notices*, 45, 2010.

T. Petricek and D. Syme. Afax: Rich client/server web applications in f#. 2007.

G. Radanne, J. Vouillon, V. Balat, and V. Papavasileiou. Eliom: tierless web programming from the ground up. 2016.

M. Serrano and V. Prunet. A glimpse of hopjs. In *International Conference on Functional Programming (ICFP)*, 2016.

T. M. Vii, K. Crary, and R. Harper. Type-safe distributed programming with ml5. In *International Symposium on Trustworthy Global Computing*. Springer, 2007.

High-Performance Web Apps through Haskell EDSLs

Abstract

We present *Aplite*, a domain-specific language embedded in Haskell for implementing performance-critical functions in client-side web applications. In Aplite, we apply partial evaluation, multi-stage programming and techniques adapted from machine code-targeting, high-performance EDSLs to the domain of web applications. We use Aplite to implement, among other benchmarks, procedural animation using Perlin noise [Perlin, 2002], symmetrical encryption and K-means clustering, showing Aplite to be consistently faster than equivalent hand-written JavaScript – up to an order of magnitude for some benchmarks. We also demonstrate how Aplite's multi-staged nature can be used to automatically tune programs to the environment in which they are running, as well as to inputs representative of the programs' intended workload.

High-performance computation in the web browser is an attractive goal for many reasons: interactive simulations and games, cryptographic applications and reducing web companies' electricity bills by outsourcing expensive computations to users' web browsers. Similarly, functional programming in the browser is attractive due to its promises of simpler, shorter, safer programs. In this paper, we propose a way to combine the two.

1 Introduction

As the rise of the web as an application platform continues unabated, we expect our browsers to be able to do more and more. Applications that were unthinkable as web applications just a few years ago – word processing, photo editing and even relatively complex computer games – are now everyday occurrences in the browser. This is hardly surprising: the speed and ease with which one can develop and deploy a web application, not having to deal with a combinatorial explosion of possible client systems and configurations, is most appealing. Equally so is the low barrier for users to try out a new application. Another compelling argument for developing for the web is the software-as-a-service business model, where software is leased on a monthly basis rather than sold. This enables updates and bug fixes to be deployed to users as they become available, rather than having to wait for the next release cycle.

JavaScript: Language of the Web Unfortunately, JavaScript, the de facto language of the web, has its share of shortcomings. While developers don't have to contend with a multitude of operating systems and system configurations when writing JavaScript, they instead must contend with a slightly smaller multitude of web browsers and JavaScript engines. Although a program written for one modern JavaScript engine will generally have the same semantics when moved to another one – in contrast to native applications – the same can not be said for performance. A program that runs at a passable speed on Google's V8 JavaScript engine may run significantly slower on Mozilla's SpiderMonkey, and vice versa. During our experiments, we observed some programs to run as much as an order of magnitude faster on V8 than on SpiderMonkey, while modifying the program to perform well on SpiderMonkey instead caused it to run significantly slower on V8.

Even if we manage to produce code that performs similarly across different browsers, we might not get the speed that we want from our JavaScript code. JavaScript can perform on par with lower-level languages under the right circumstances. The *ASM.js* low-level subset of JavaScript, for instance, can perform well enough to be used to implement demanding 3D games [Zakai, 2013]. However, ASM.js is intended as a compilation target, and is thus not well suited to being written by humans. Disciplined use of plain JavaScript can give some of the benefits of ASM.js, but JavaScript contains several constructs that are highly problematic from an optimisation point of view: higher order functions, the *eval* function, reflection and reliance on weak, dynamic typing can all have adverse effects on JavaScript performance, to name a few. A single performance misstep may lead to subpar performance in *all* browsers. Optimisation-unfriendly code, in addition to being slow in and of itself, may in some cases cause deoptimisation of *all surrounding code* by invalidating assumptions needed for efficient compilation of a given function [Antonov, 2015]. The untyped nature of JavaScript also means that the programmer needs to manually keep track of the types of all values their programs operate on, adding manual type checks and additional testing to compensate for a lack of static guarantees. Being forced to write fast, clever, low-level code only exacerbates this problem.

Considering that we generally do not trust developers to produce type-safe code through sheer discipline – hence the use of type checkers – it would seem unwise to trust those same developers to be disciplined enough to produce code which is guaranteed to avoid performance pitfalls. Doubly so when the performance pitfalls differ depending on the user's choice of web browser.

Functional Languages to the Rescue? As functional programmers, can we use functional programming languages to solve the aforementioned problems? Most major functional languages can now be compiled to JavaScript, including some designed specifically for the web. Haskell through GHCJS [Nazarov et al., 2015], Haste [Ekblad, 2015a] or UHC [Dijkstra et al., 2013], OCaml through Js_of_ocaml [Vouillon and Balat, 2014] are all instances of the former, while Elm [Czaplicki, 2012] and PureScript [Freeman, 2015] are instances of the latter.

However, programs written in many of these languages can be quite slow when executed in the browser, especially for computationally heavy tasks. Not only can the JavaScript engine's garbage collector be a poor match for functional programs, but the JavaScript-targeting compilers also generally lack the hundreds of person-years of effort that went into their native cousins, and many of the tricks used to speed up execution of functional languages on stock hardware do not apply to JavaScript. Additionally, JavaScript code compiled from functional languages often make heavy use of constructs, higher order functions in particular, that are hard for JavaScript engines to fully optimise [Ekblad, 2015a].

EDSLs to the Rescue! We don't want to force developers to manually enforce type-correctness and disciplined, performance-oriented programming, as is needed when using JavaScript. However, as poor performance is one of the major problems we would like to solve, we can also not accept trading convenience and safety for even *lower* performance, as with functional languages compiled to JavaScript.

Instead, we look to *EDSLs* – embedded, domain-specific languages – for a solution. The idea of using domain specific languages to achieve high performance is not new: in order to make a language excel in a specific domain, its syntax and types are restricted so that only problems in the given domain are expressible. Without having to support the generality of a general-purpose programming language, the compiler is free to focus on optimising for the given domain, being able to make assumptions about programs that would otherwise be invalid. Meanwhile, less performance-sensitive code may be written in the slower, more expressive host language, giving us either performance or convenience on a case by case basis.

A type-safe, domain-specific language for computations which are highly optimisable can elegantly solve the aforementioned problems with resorting to hand-written JavaScript. Furthermore, embedding the language in a higher-level host language yields additional benefits. EDSL code may take advantage of the host language for advanced meta-programming capabilities [Axelsson et al., 2010]. This embedding also elegantly enables run-time specialisation of EDSL code: being compiled on-line by the host language program itself, EDSL code may be specialised to user input during run-time, possibly yielding more efficient code; a technique commonly known as partial evaluation [Futamura, 1999]. The host program may take advantage of its knowledge of the browser it is currently executing in to tune the code generator to produce the most efficient code possible for the current environment.

It should be noted that EDSLs for the browser is not an all-or-nothing proposition. Low-level, performance-focused EDSLs are likely to yield the best results performance-wise, while using Haskell without any EDSLs at all may give programmers more expressive power at the cost of performance. However, one might also consider taking the middle road: a higher-level, web-focused EDSL might yield a smaller performance benefit than a lower-level one, but allows the programmer to still work at a comparatively higher level of abstraction. One might also consider mixing EDSLs of different layers of abstraction, giving programmers more freedom to choose the tool for the task at hand. This approach is discussed further in section 6.

Our Contribution This paper makes the following contributions:

- In sections 2 and 4, we describe the *Aplite* multi-backend EDSL, targeting a low-level subset of JavaScript based on *ASM.js*, compiled and executed on demand in the user's browser. Aplite constitutes a special case of multi-stage programming, allowing specialising code to the browser as well as to user input. With these properties, Aplite is designed to mitigate the performance problems of computationally heavy web applications.

- In section 3, we demonstrate how the multi-staged, multi-backend nature of Aplite can be used to automatically select at run-time the most efficient backend for any given program in the current browser environment, and to seamlessly integrate Aplite code into Haskell programs, using type families to separate pure Aplite kernels from impure ones allowing both to be imported using the same interface.

- In section 5, we implement several programs, including procedural animations using Perlin noise, symmetric encryption and K-means clustering, using Aplite, and show consistent performance improvements over equivalent, hand-written JavaScript programs, up to an order of magnitude for some benchmarks.

2 Aplite: A High-Performance JavaScript EDSL

The Aplite language is implemented as a strongly typed DSL embedded in Haskell. More specifically, it is intended for use with web-targeting Haskell dialects such as GHCJS, Haste or UHC. It has two main design

```
-- Expression language
type CExp a
instance Num a ⇒ Num (CExp a)
instance Bits a ⇒ Bits (CExp a)

-- Comparisons
(#&&) :: CExp Bool → CExp Bool → CExp Bool
(#==) :: JSType a ⇒ CExp a → CExp a → CExp Bool
...

-- Numerics
sqrt_ :: JSType a ⇒ CExp a → CExp a
...

-- Command language
type Aplite a
instance Monad Aplite

-- References
type Ref a
initRef   :: CExp a → Aplite (Ref a)
getRef    :: Ref a → Aplite (CExp a)
setRef    :: Ref a → CExp a → Aplite ()
modifyRef :: Ref a → (CExp a → CExp a) → Aplite ()

-- Arrays
type Arr i e
newArr    :: Ix i
          ⇒ CExp i → Aplite (Arr i e)
getArr    :: Ix i
          ⇒ Arr i e → CExp i → Aplite (CExp e)
setArr    :: Ix i
          ⇒ Arr i e → CExp i → CExp e → Aplite ()
modifyArr :: Ix i
          ⇒ Arr i e → CExp i → (CExp e → CExp e)
          → Aplite ()

-- Control flow
data Border i = Incl i | Excl i
for :: Integral i
    ⇒ (i, Int, Border i)
    → (CExp i → Aplite ())
    → Aplite ()
while :: Aplite Bool → Aplite () → Aplite ()
iff :: CExp Bool → Aplite () → Aplite ()
    → Aplite ()
ifE :: CExp Bool → Aplite (CExp a) → Aplite (CExp a)
    → Aplite (CExp a)
```

Figure 17: Constructs of the Aplite language

```
square :: CExp Int → Arr Int Double → Aplite ()
square len arr =
  for(0, 1, Excl len) $ \i →
    modifyArr arr i (**2)
```

Figure 18: An Aplite example: squaring an array of numbers

```
squareArray :: Int → IOUArray Int Double → IO ()
squareArray = aplite square

main :: IO ()
main = do
  arr ← newListArray (0, 9) [1..10]
  squareArray 10 arr
  getElems arr >>= print
```

Figure 19: Calling Aplite from Haskell

goals: to allow numeric computations to be expressed in a way that is efficiently executable across JavaScript implementations and to easily integrate with high-level Haskell code. It effectively creates a hybrid programming environment, where control flow and event handling may be expressed in high-level Haskell, while performance-critical computations may be outsourced to highly efficient Aplite kernels. The language is deeply embedded and intended to serve as a high-performance core on which to base higher-level functional abstractions. Thus it takes on a role similar to that of the deep embedding in Axelsson's and Svenningsson's work on combining deep and shallow embedded DSLs [Svenningsson and Axelsson, 2012]. This makes the language easily extensible: rich features may be added on top, compiling down to a relatively small and simple core. The code generator and user interface of the language may thus be extended and improved independently of each other.

Due to the first design goal – restricting ourselves to efficient code only – the language is quite simple. It allows programmers to express bitwise operations over integers, arithmetic operations over floating point and integer values, and access to mutable arrays, references, conditional statements and loops in an *Aplite* monad, which is reified using the technique presented by Svenningsson and Svensson [Svenningsson and Svensson, 2013].

The list of supported constructs is heavily guided by what constructs are supported by the low-level, highly efficient *ASM.js* subset of JavaScript, which is covered in more detail in section 4.1. Lambda abstractions and general recursion are not supported by Aplite's syntax. Instead we rely on Haskell functions to break up Aplite programs in manageable units, inlining all applications. The pros and cons of this approach are explored further in section 6.

Figure 18 shows a simple example of an Aplite program, squaring the contents of an array.

Unlike many EDSLs, Aplite programs are not compiled beforehand, nor are they complete applications on their own. Aplite programs have no way of communicating with the outside world, being limited to a strictly computational role. Instead, Aplite programs are *imported* into the host program using the *aplite* bridging function. This function takes as its input an Aplite function over n arguments $a_1 \rightarrow ... \rightarrow b$, and returns a native Haskell function over n arguments $a'_1 \rightarrow ... \rightarrow b'$, where the Haskell types $a'_{1..n}$ and b' are isomorphic to the Aplite types $a_{1..n}$ and b respectively. Figure 19 shows how functions written using Aplite can be called from the host program. A mutable array containing the numbers 1 to 10 is created, passed to the *square* function from figure 18 which updates the array to contain the squares of its original contents, converted to a list and printed.

Note how the type of the imported *squareArray* function differs from that of *square*: the *Aplite* monad is replaced by *IO*, the *Word32* is no longer locked up in Aplite's *CExp* expression type, and the array has become a standard unboxed, mutable Haskell array. On import, the *aplite* function automatically marshals arguments and return values between Haskell and Aplite, provided that the type of the Aplite function is isomorphic to that of the import. Attempting to import an Aplite function under an incompatible type signature yields a Haskell type error at compile-time. Compilation of any well-typed Aplite program is guaranteed to succeed.

This automatic conversion of arguments and return values not only makes it easy to use Aplite in a higher-level Haskell control program, but also ensures that values can not flow unchecked, by closure or otherwise, between Haskell and Aplite code. This property lets us import certain Aplite functions as though they were pure. While any Aplite function may be imported into the *IO* monad, functions which are guaranteed to contain no observable effects can safely be imported as pure Haskell functions as well, as in the example given by figure 20. A function is considered safe to import purely if it does not accept any mutable arrays as arguments. Mutable references are disallowed in imports in general, and thus do not affect the purity of an import. Returning a mutable array is perfectly safe however, as the ban on mutable arguments ensures that any such array must have been created within the function. This means that even if the function were to have mutated the array, nobody else would have had a

```haskell
fib :: Int → Double
fib = aplite $ \x → do
  r1 ← initRef 0
  r2 ← initRef 1
  for (1, 1, Excl x) $ \i → do
    x1 ← getRef r1
    x2 ← getRef r2
    setRef r1 x2
    setRef r2 (x1 + x2)
  getRef r1
```

Figure 20: Calculating the *n*th Fibonacci number with Aplite

reference to the array to observe the mutation taking place. Attempting to import a potentially effectful function as a pure function yields a type error. The type-level implementation of these restrictions is further discussed in section 3.

Aplite takes advantage of Haskell's non-strict semantics to provide on-demand compilation and loading of functions. Up until the point where evaluation of *squareArray* is actually forced, it is merely a chunk of syntax tree, sitting around awaiting eventual compilation. This means that not only can we defer the cost of compilation until we know that compiling a function would be productive, but functions which end up not being used can be pruned during the compiler's normal dead code elimination pass, lowering the amount of data that needs to be sent to client browsers before execution can begin.

Figure 17 gives an overview of the constructs of the Aplite language. The *JSType* type class contains all permissible base types of the language: signed and unsigned integers, floating point numbers, and booleans. The *Border* construct describes a border value for a for-loop: a border value of *Incl n* indicates that a for-loop with loop variable i will execute while $i <= n$, while *Excl n* indicates that the loop will execute while $i < n$.

3 Interfacing with Haskell

An Aplite function is imported into its host programs by passing it to the *aplite* function and assigning the result an appropriate type signature, giving the type at which we would like to use the function in our Haskell host program. A corresponding Aplite-level type signature is derived from this Haskell-level type signature in order to check that it is actually possible to import the function at the type that we want. Deriving the Aplite-level type from the Haskell-level type instead of the other way around ensures

```haskell
type family ApliteSig a where
  ApliteSig (a → b) = ApliteArg a → ApliteSig b
  ApliteSig a       = ApliteResult a

type family ApliteArg a where
  ApliteArg Double        = CExp Double
  ApliteArg Int32         = CExp Int32
  ...
  ApliteArg (IOUArray i e) = Arr i e

type family ApliteResult a where
  ApliteResult (IO (IOUArray i e)) = Aplite (Arr i e)
  ApliteResult (IO ())             = Aplite ()
  ApliteResult (IO a)              = Aplite (CExp a)
```

Figure 21: Deriving Aplite types from Haskell equivalents

that we do not have to explicitly type Aplite programs before passing them to *aplite* – the appropriate type will be inferred from the Haskell-level type, ambiguity avoided.

3.1 Deriving Aplite Types

The process of deriving an Aplite-level type from its Haskell-level equivalent is relatively straightforward. We recurse through arguments of the Haskell-level type, changing the types of the arguments into their Aplite equivalents. Figure 21 shows the procedure implemented using closed type families [Eisenberg et al., 2014]. The *ApliteArg* type family contains conversions for all supported argument types, while the *ApliteResult* family covers all supported return values. The *CExp* type is the type of Aplite expressions.

However, this only covers imports in the *IO* monad. In order to support pure imports of "safe" functions as well, we must extend this scheme to cover the notion of pure and impure functions, and somehow guard against impure functions being imported purely – recall that a function is considered impure if it accepts a mutable array as an argument.

We do this by introducing another type family – *Purity* – which looks at the return type of any function type to determine if it is of the form *IO a*, in which case the function is *Impure*, or some other type *a*, which means that the function is *Pure*. We then extend the *ApliteArg* family with an extra parameter *p*, giving the purity of the function the argument belongs to. By simply omitting the type instance for the case where the argument is a mutable array and *p* is anything but *Impure*, we ensure that type checking of such functions fail.

```haskell
data Pure
data Impure

type family Purity a where
  Purity (a → b) = Purity b
  Purity (IO a)   = Impure
  Purity a        = Pure

type family ApliteSig a where
  ApliteSig (a → b) = ApliteArg a (Purity b)
                          → ApliteSig b
  ApliteSig a       = ApliteResult a

type family ApliteArg a p where
  ApliteArg Double         p      = CExp Double
  ApliteArg Int32          p      = CExp Int32
  ...
  ApliteArg (IOUArray i e) Impure = Arr i e

type family ApliteResult a where
  ApliteResult (IO (IOUArray i e)) = Aplite (Arr i e)
  ApliteResult (IO ())             = Aplite ()
  ApliteResult (IO a)              = Aplite (CExp a)
  ApliteResult (IOUArray i e)      = Aplite (Arr i e)
  ApliteResult a                   = Aplite (CExp a)
```

Figure 22: Aplite type derivation extended with purity

This slightly tricky part done, all that is left is to extend the *ApliteResult* family of permissible return types with the non-IO cases, and we're done. Figure 22 updates the code from figure 21, extending it to support pure imports as well as impure.

3.2 From Function to Concrete Program

While deriving the Aplite type that corresponds to an import signature is all well and good, we also need a corresponding value-level transformation. Figure 23 shows the implementation of this transformation.

In order to be able to import Aplite functions, we must first reduce them from lambda expressions to concrete programs. Using the *square* function from figure 18 as an example, we must apply it to some *CExp Length* and some *Arr Index Double* to obtain a fully saturated value of type *Aplite ()*, which we can then compile into our intermediate representation.

We do this by recursing over the arguments of the function. For each argument, we generate a unique name *n* and store it in a list of arguments.

```
class Export f where
  type Result f
  export :: Id → [(Type, Id)] → f → ([(Type, Id)], Aplite ())

instance Export (JSType a, Export b) ⇒ Export (CExp a → b) where
  type Result (CExp a → b) = Result b
  export n args f = export (succ n) args' f'
    where
      argType = jsType (undefined :: CExp a)
      argName = "arg" ++ show n
      args'   = (argType, argName) : args
      f'      = f (varExp argName)

instance ReturnValue a ⇒ Export (Aplite a) where
  export _ args f = (reverse args, f >>= return_)
```

Figure 23: Exporting Aplite functions

We then construct an Aplite expression representing a variable *v* with the name *n*, using the *varExp* helper function, and apply the function being exported to *v*, resulting in a function with one fewer argument. After the function has been fully applied in this way we bind the resulting, now saturated, Aplite program with a function *return_*, which produces an actual, function-terminating return statement in the resulting code (as opposed to monadic return, which does not cause termination), and return the final program along with its list of arguments.

In figure 18, the previously mentioned *return_* function resides in a *ReturnValue* type class, as it needs to be overloaded for Aplite's *CExp* expression type as well as for arrays. The type of each argument in the intermediate representation, needed by the code generator, is calculated using the *jsType* function which resides in the type class by the same name. For brevity's sake, we will not give the full implementation of either type class here; the informal descriptions given in this paragraph will have to suffice.

3.3 Compilation and Loading

After being thus transformed, the finished Aplite program and its argument list are passed to the code generator, further discussed in section 4, where it is compiled into a string containing a JavaScript function representing the program. While a string is great for inspecting the generated code, it would normally not do us much good when it comes to actually loading and executing it. In our case, however, we will use the *Haste.Foreign* [Ekblad, 2015b] foreign function interface to load the generated code.

Haste.Foreign is a lightweight foreign function interface designed specifically for JavaScript-targeting Haskell dialects. Using JavaScript's native *eval* function as its backend, it allows importing raw strings of JavaScript code into Haskell as full-fledged functions. Much like our *aplite* function imports an Aplite function under whatever – matching – type signature we give it, Haste.Foreign provides a *ffi* function which imports a string of JavaScript under a given type signature. Given that JavaScript is a dynamically typed language, there is no requirement that the JavaScript string is somehow well-typed with respect to the type at which it is imported. The only restriction placed on imported functions is that their arguments and return value are marshallable between Haskell and JavaScript, and that functions are imported in the *IO* monad, reflecting the fact that JavaScript code may perform arbitrary side-effects.

While the general safety of importing dynamic strings of code at arbitrary types can certainly be questioned, it fits our purposes quite nicely. There is, however, one bit of impedance mismatch: Haste.Foreign's restriction to imports in the *IO* monad does not square well with our wish to import pure Aplite functions when provably safe. This means that we can not simply import our compiled Aplite functions using the type at which we intend to call it. The solution is to derive yet another type for our Aplite program: its *FFI type*, which we will use to import the generated JavaScript. This type is quite simple; if the Haskell type of an Aplite function is $a_1 \rightarrow ... \rightarrow IO\ b$, then its FFI type remains $a_1 \rightarrow ... \rightarrow IO\ b$. If its FFI type is $a_1 \rightarrow ... \rightarrow b$ for any other b, then its import type becomes $a_1 \rightarrow ... \rightarrow IO\ b$.

After importing our function through Haste.Foreign at its FFI type, we must now somehow strip away the final *IO*, if the user requested the import to be pure. We do this by recursing through the arguments of the function, leaving them untouched, until we reach the *IO* computation at the bottom of the type. There, we apply *unsafePerformIO* to the computation, allowing the function to escape the *IO* monad. This use of *unsafePerformIO* is completely safe, as the *ApliteSig* type family ensures that attempting to import an Aplite function with observable effects purely results in a type error.

Figure 24 shows the derivation of an Aplite function's FFI type, and the release of a pure imported function from the IO monad. Note the use of an additional parameter *p* in the *Escaped* type class to indicate the purity of the function being examined.

```
type family FFISig a where
  FFISig (a → b) = a → FFISig b
  FFISig (IO a)   = IO a
  FFISig a        = IO a

class EscapeIO a p where
  type Escaped a p
  escapeIO :: p → a → Escaped a p

instance EscapeIO b p ⇒ EscapeIO (a → b) p where
  type Escaped (a → b) p = a → Escaped b p
  escapeIO p f = \x → escapeIO p (f x)

instance EscapeIO (IO a) Pure where
  type Escaped (IO a) Pure = a
  escapeIO _ = unsafePerformIO

instance EscapeIO (IO a) Impure where
  type Escaped (IO a) Impure = IO a
  escapeIO _ = id
```

Figure 24: FFI types and escaping from the IO monad

3.4 All Together Now

Putting together the pieces of the puzzle, we end up with several restrictions on both the Aplite and Haskell-level types of the functions we wish to import:

- The function's *Aplite type* must be an instance of *Export* so that we may gather up its arguments and convert it into a concrete Aplite program.
- The function's *FFI type* must be an instance of Haste.Foreign's *FFI* type class, to ensure that programs are actually importable.
- The function's *FFI type* must be an instance of the *EscapeIO* type class, allowing us to let pure imports out of the *IO* monad.
- The function's *Haskell type* must be equivalent to the result of escaping its *FFI type* from the *IO* monad.

With these restrictions now gathered in one place, all that's left is to glue together the pieces described throughout this section. Figure 25 shows the complete implementation of the *aplite* function, broken down into type-annotated steps for readability. Note the use of the *JSString*

```
type ApliteExport a =
  ( Export (ApliteSig a)
  , Haste.Foreign.FFI (FFISig a)
  , EscapeIO (FFISig a) (Purity a)
  , a ~ Escaped (FFISig a) (Purity a)
  )

aplite :: ∀a. ApliteExport a ⇒ ApliteSig a → a
aplite prog = escapeIO (undefined :: Purity a) prog4
  where
    prog2 :: ([(Type, Id)], Aplite ())
    prog2 = export 0 [] prog

    prog3 :: JSString
    prog3 = CodeGen.generate prog2

    prog4 :: FFISig a
    prog4 = Haste.Foreign.ffi prog3
```

Figure 25: The *aplite* function revealed

type, representing a JavaScript-native string, to avoid costly conversion between Haskell's *String* type and the JavaScript's more efficient native representation that the *eval* function underlying Haste.Foreign demands as its input. Note also that *ApliteExport* here is a constraint, and as such requires the *ConstraintKinds* Haskell extension.

4 Code Generation

Aplite draws its inspiration from the Feldspar [Axelsson et al., 2010] high-performance DSP and array processing EDSL. In fact, it is based on *imperative-edsl* [Axelsson, 2015], a lightweight Feldspar offshoot intended to provide an efficient low-level base language on top of which higher-level functional abstractions may be built in a resource-aware manner.

While imperative-edsl compiles down to a symbolic representation of the C language, this is not ideal for our use case. C supports some functionality that can not be implemented efficiently in JavaScript – most notably unstructured jumps – and is in general a larger language than what we need from an intermediate representation. For Aplite, we have modified imperative-edsl to instead use a typed, specialised subset of JavaScript, where each sub-expression is annotated with its type and where only efficient language constructs are representable. We deem a language construct to be efficiently representable if it is translatable into a JavaScript construct which can in turn be translated into some efficient machine code construct by the JavaScript engine. In short, this includes:

- Local, mutable variables
- Logic, arithmetic and bitwise operators
- If-, for- and while-statements
- Array reads and writes

The intermediate language also has a very limited type system: only signed and unsigned integers, double precision floating point numbers, and arrays over said types are allowed. In general, the intermediate language corresponds closely to an abstract representation of ASM.js.

As one of the main motivations for this work is the inability of JavaScript code to perform consistently across JavaScript engines, Aplite supports multiple backends and aims to enable developers to choose the backend that best suits their particular situation. The backend used to compile a particular function may be configured by replacing the *aplite* function seen in figures 19 and 20 by the *apliteWith* function, which accepts an additional backend configuration parameter as input. Since this parameter is configurable at run-time – as this is when Aplite programs are compiled – the developer may also take user input and the browser environment in which their application is currently executing into account when choosing how to optimise their performance-critical functions. Section 4.3 describes how selection of the optimal backend for any given Aplite program can be automated.

Additionally, having multiple backends can be greatly useful in ascertaining the correctness of the backends, as each backend serves as an oracle for each other backend. Testing the backends against each other provides a cheap way to create a large number of tests with relatively little overhead. While such a testing regimen is not enough to conclude that both backends are completely free from errors, it provides a relatively solid defence against regressions in either backend.

We have implemented two code generators targeting different flavors of JavaScript: plain, but low-level, JavaScript, and ASM.js.

4.1 ASM.js

ASM.js is a subset of the JavaScript language which has a one-to-one mapping to machine code, chosen to act as an efficient compilation target for JavaScript-targeting compilers [Herman et al., 2014]. When a supported

```
var squareHeapModule = (function(stdlib, ffi, heap) {
  "use asm";

  var imul = stdlib.imul;
  var intHeap = new stdlib.Int32Array(heap);

  function squareHeap(len) {
    len = len|0;
    var i = 0;
    var tmp = 0;
    for(i = 0; (i|0) < (len|0); i = (i + 1)|0) {
      tmp = intHeap[(i << 2) >> 2];
      intHeap[(i << 2) >> 2] = imul(tmp, tmp);
    }
  }

  return ({squareHeap: squareHeap});
})(Math, null, new ByteArray(0x1000));
```

Figure 26: Squaring *n* elements on the heap with ASM.js

web browser encounters such JavaScript, it more or less bypasses its normal interpretation pipeline and instead compiles the code straight down to machine code. ASM.js is cleverly designed to be backwards-compatible: any valid ASM.js program is also a valid program in plain JavaScript. Thus, any ASM.js program will work in any modern browser, with the only drawback that unsupported browsers will run the code significantly more slowly.

Unlike normal JavaScript, ASM.js is typed, using standard operators to act as type annotations: a $+$ prefix indicates that a value is a double precision floating point number, a $|0$ suffix – bitwise *or* by zero – indicates a signed integer, and a $>>> 0$ suffix – logical right shift by zero – indicates an unsigned integer. These annotations must be sprinkled liberally throughout the program: JavaScript's only concept of numbers is its double precision floating point *Number* type, and the type annotations are chosen to force their result to obey the semantics of the type they represent. Performing bitwise operations on a JavaScript number forces it to behave as a 32-bit signed integer, except for logical right shift which forces unsigned 32-bit integer behaviour instead. Hence their selection as type annotations. Each subexpression must thus carry the appropriate annotation to avoid reverting back to standard behaviour for JavaScript numbers.

ASM.js programs have severe restrictions on the functionality they can use. Only value types – numbers – may be used, so that memory may be allocated exclusively on the stack, avoiding the need for garbage collection. Any more complex, or persistent, data must be stored in an explicit heap represented as a JavaScript *typed array* – an efficient, unboxed representation of a raw string of bytes – the size and location of which are given when an ASM.js module is compiled. The heap is wrapped in one or more views, which lets programs access its contents as elements – integral and floating point numbers of different sizes – rather than as a string of bytes. Indices into these views must be bit-shifted right by two or three, depending on the element size of the view being used. Bit-shifting an index into a 32-bit integer view of the heap to the right by two divides it by four, giving the byte offset at which the indexed value resides. ASM.js uses this information to retrieve the address at which the given element can be found – the index before bit-shifting is the byte index of the element – which is needed when addressing memory at the machine level to which ASM.js compiles.

ASM.js code may only call functions that are either available in a designated standard library object – essentially a tiny subset of libc – or specified at compile-time in a special FFI object. In order to be recognised by the browser and accordingly optimised, ASM.js code must reside in a *module* – a concept which in JavaScript generally refers to a function which when executed exports an object containing one or more library functions – which has a very particular layout. References to external functions through the FFI or standard library objects must be copied into local variables, to ensure that they are not modified from the outside during execution. Any functions containing ASM.js code must be declared – locally, again to avoid outside modification – and returned as part of an object containing the interface to the module. Figure 26 gives a complete example of an ASM.js module.

These restrictions make communication between ASM.js and other JavaScript code cumbersome. This is not necessarily a drawback for its intended use case however, as it is mainly intended as a compilation target for whole applications, rather than small snippets of performance-critical code in a larger JavaScript application.

The intermediate language to which Aplite is compiled before the code generation step was essentially designed around the needs of an ASM.js code generator, hence its insistence on type annotations at every subexpression. The ASM.js backend is thus essentially a pretty-printer for the intermediate format. It also produces the module structure described above and some scaffolding essential to setting up the ASM.js environment. Recall that all ASM.js programs must have a single heap, specified at compile-time. Since we cannot substitute this heap for something else at run-time, any array data passed into an Aplite function must be copied

```
var squareHeapModule = (function() {
  function squareHeap(len, arr) {
    var i, tmp;
    for(i = 0; i < len; ++i) {
      tmp = arr[i];
      arr[i] = Math.imul(tmp, tmp);
    }
  }
  return ({squareHeap: squareHeap});
})();
```

Figure 27: Squaring *n* elements using "plain" JavaScript

into the function's heap. When the function returns, that same data must be copied back *out* of the heap and into its original array again, as any mutation of input arrays must be observable from the calling program.

These array-related caveats make the ASM.js backend relatively unsuited for functions which work on large arrays: even should the ASM.js code generator be the fastest one for the task, the cost of copying millions of elements into and out of arrays quickly becomes prohibitive.

4.2 Plain JavaScript

The plain JavaScript backend is quite similar to the ASM.js backend, but dispenses with the elaborate module declarations and much of the line noise of the type annotations. Some annotations are still necessary – bitwise *or*ing or right-shifting by zero after arithmetic to avoid overflow for instance – but quite a few can be done away with. Most crucially, the JavaScript backend does not use an explicit heap. While it does use the same typed arrays that underpin the ASM.js heap, it uses an unbounded number of separate array objects instead of squeezing all the data into one big array. This means that we no longer need to copy data into and out of the heap: data can be shared directly between Aplite programs and unboxed Haskell arrays. Figure 27 gives an equivalent example, in the "plain" JavaScript flavor, to the ASM.js module in figure 26.

This gives the plain JavaScript backend a distinct advantage when working with potentially large arrays. This format is also preferable for web browsers which do not support ASM.js: while supported web browsers ignore the excessive bitwise operations-as-type-annotations of ASM.js at run-time, simply using them to guide compilation down to code which implements the same functionality in hardware, unsupported browsers may in the worst case be completely unable to optimise the operations away, ending up performing a lot of unnecessary work. The output of the plain JavaScript backend, containing less casts, annotations and strange hoops for interpreters to jump through in general, may thus be more palatable to some browsers. Even where ASM.js is supported, the plain JavaScript backend may be preferable. In our experiments, several benchmarks fared better with the plain JavaScript backend than with the ASM.js one in both browsers, despite both being ASM.js-aware.

4.3 Automatic Backend Selection

Embedded DSLs make experimentation and exploratory programming significantly easier than when writing the target code by hand: small changes to the source code yield large changes in the target program, and switching backends altogether may yield dramatic effects on performance. However, figuring out the appropriate backend for every situation is a non-trivial task in its own right: web browsers are frequently updated, and, as discussed in section 5, the optimal backend also depends heavily on the program under compilation.

Thanks to Aplite's multi-staged nature, we can relieve the programmer of this burden by providing a means to automatically select the appropriate backend for the program under consideration and the present execution environment. To this end, we add a new function *apliteSpec* which, in addition to a function $f :: a$, which contains an indirection to a compiled Aplite function, returns a specialisation handle $h :: \mathit{SpecHandle}\ a$ for f. A specialisation handle contains the abstract syntax tree of an Aplite function, allowing it to be recompiled at need, and a reference to the compiled Aplite function to which f is an indirection.

The handle h may then be passed to a function *specialize* along with a benchmarking function $b :: a \to \mathit{IO\ Time}$, which may perform any required benchmarking setup and measure the execution time of its argument in whichever manner the developer finds appropriate. Then, *specialize* recompiles the syntax tree contained in h with both backends, applies b to each resulting function, and replaces the underlying function of f with whichever implementation turned out to be the fastest one. Figure 28 gives an implementation of this scheme, and figure 29 shows an example of how it may be used by the developer. In the example, we create an array to use as test data, apply the function we want to benchmark once, to ensure that the initial overhead of lazy compilation doesn't affect the timings, and then time a second application of the function to the test data. The *specialize* function then uses the obtained timing information to determine which backend is faster.

```
type Time = Double

data SpecHandle a = SpecHandle
  { funcAst  :: ApliteFunc
  , funcCode :: IORef Dynamic
  }

apliteSpec :: ∀a. ApliteExport a ⇒ ApliteSig a → (a, SpecHandle a)
apliteSpec = unsafePerformIO $ do
  r ← newIORef (apliteWith ct prog :: a)
  return (unsafePerformIO $ readIORef r, SpecHandle (compileToAST prog) r)

specialize :: ∀a. ApliteExport a
           ⇒ SpecHandle a
           → (a → IO Time)
           → IO ()
specialize SpecHandle{..} bench = do
    tf ← bench f
    tg ← bench g
    writeIORef funcCode $ if tf ⩽ tg then f else g
  where
    f = apliteFromAST defaultTuning funcAst :: a
    g = apliteFromAST asmjsTuning funcAst :: a
```

Figure 28: The *apliteSpec* and *specialize* functions

This method generalises nicely to any number of backends and code tunings. By letting the user provide a list of backends to consider in addition to the current arguments of *specialize*, the user could also be allowed to guide specialisation by ruling out parts of a possibly large number of backends and tunings from the start.

Automatic backend selection is implemented here as an effectful operation rather than as a pure function from an unspecialised Aplite program to a specialised one, to give the programmer more control. Depending on the number and nature of the functions to specialise and their inputs, specialisation may take some time to perform. The programmer may thus want to be able to control when, and if, this specialisation happens – for instance to display a loading screen while specialisation is underway. Having specialisation as a pure function would mean that the newly specialised functions would either have to be created on the top level, denying the programmer this control, or be threaded throughout the entire application.

```
-- Time the execution of a computation
time :: IO a → IO Time

square :: Word32 → IOUArray Index Double → IO ()
(square, specSquare) = aplite $ \len arr →
  for(0, 1, Excl len) $ \i →
    modifyArr arr i (**2)

main :: IO ()
main = do
  displayLoadScreen
  specialize specSquare $ \square → do
    arr ← newListArray (0, 9) [1..10]
    square 10 arr
    time $ square 10 arr
  startApplication
```

Figure 29: Using automatic backend selection

5 Performance Evaluation

To evaluate the performance of the Aplite language, we bring out a range of benchmarks, selected to cover Aplite's performance across a wide range of tasks: prime factorisation, procedural animation using Perlin noise, K-means clustering, matrix multiplication, symmetric encryption and CRC32 checksums. While some of our experiments did show Aplite-generated JavaScript outperforming native, GHC-compiled Haskell code, the focus of Aplite is to augment the performance of client-side JavaScript and browser-targeting Haskell code. For this reason, we compare the performance of Aplite to that of hand-written JavaScript and the web-targeting Haste implementation of Haskell, rather than server-targeting native Haskell.

Methodology Each program was implemented using Aplite and compared to an equivalent hand-written JavaScript version of the same program.[1] For illustrative purposes, the Aplite implementation of the *matrix* benchmark is shown in figure 30, and its hand-written JavaScript counterpart is shown in figure 31.

The benchmark programs are briefly described below.

- *crc32* - calculating the CRC32 checksum of 100 MB of data. The data is passed to the Aplite function as a mutable array, which stresses

[1] All benchmarks can be downloaded from https://github.com/valderman/aplite-benchmarks.

the ASM.js backend disproportionately due to the copying issues described in section 4. This benchmark uses the highly optimised standard implementation of CRC32 from the SheetJS [2014] open source library for its JavaScript version.

- *K-means* - K-means clustering of 100,000 random points into five clusters in two dimensions. The points and initial cluster centre points are uniformly distributed across the plane. The K-means benchmark is specialised to the number of clusters using the partial evaluation we get for free by compiling our language at run-time. Similarly to *crc32*, points and clusters are passed to the Aplite function in mutable arrays, exercising the ASM.js backend quite significantly.

- *perlin* - procedurally generated, infinite animation of two-dimensional clouds using Perlin noise. The benchmark measures the time taken to generate 10 200x200 pixel frames of the animation, calculated from the average frame rate of the animation.

- *factors* - a naïve algorithm for finding the prime factors of a number: factors are calculated by iterating through all odd numbers up to the square root of the target number. The benchmark measures the time the algorithm takes to figure out that 5,467,154,436,746,477 is a prime. This benchmark only uses arrays to store found factors and spends most of its time performing the actual factorisation, making it an indicator of performance over simple, arithmetic functions.

- *matrix* - multiplication of two 600x600 element matrices. The handwritten JavaScript version of this benchmark uses the same typed, unboxed arrays as Aplite produces. Modifying the benchmark to instead use the slightly more idiomatic standard JavaScript arrays increases its execution time by about 10 percent. As the matrices are passed to Aplite as arrays, totalling almost a million elements, this benchmark triggers the array copying in the ASM.js backend much like the *crc32* benchmark.

- *xtea* - 4 MB of data is encrypted using the XTEA block cipher in CBC mode. While XTEA has known weaknesses and is thus not suitable for production use, it is simple to implement and verify, and it contains many of the same structures used in stronger ciphers.

The Aplite programs and the non-Aplite Haskell programs were compiled using version 0.5.4.2 of the Haste compiler, with the `--opt-all` flag, enabling all optimisations. All programs were run 10 times using Chrome

```
matMult :: Word32 → Mat → Mat → Mat → IO ()
matMult = apliteWith TUNING $ \m m1 m2 out → do
  for (0, 1, Excl m) $ \i → do
    for (0, 1, Excl m) $ \j → do
      sumRef ← initRef 0
      for (0, 1, Excl m) $ \k → do
        x ← getArr m1 (i*m+k)
        y ← getArr m2 (k*m+j)
        modifyRef sumRef (+(x*y))
      getRef sumRef >>= setArr out (i*m+j)
```

Figure 30: The *matrix* benchmark in Aplite

```
function mult(m, m1, m2, out) {
    for(var i = 0; i < m; ++i) {
        for(var j = 0; j < m; ++j) {
            var sum = 0;
            for(var k = 0; k < m; ++k) {
                sum = m1[i*m+k]*m2[k*m+j];
            }
            out[i*m+j] = sum;
        }
    }
}
```

Figure 31: The *matrix* benchmark in JavaScript

50.0 and Firefox 45.1 under Debian GNU/Linux, on a Core i5 6300U machine with 16 GB of RAM, and the median run time for each program calculated. Before each timed run, the test data was loaded into memory and the Aplite programs compiled and executed once, to ensure that test data generation and compilation times did not interfere with the test results. Some Haskell programs are problematic to implement in a way that admits a fair comparison with hand-written JavaScript and the Aplite backends, and were thus not measured implemented in Haste Haskell – an instance of this being the *factors* benchmark which relies heavily on an efficient implementation of *fmod*, which is currently not provided by Haste. In light of the remaining benchmarks and the fact that Haskell programs are almost invariably slower than their hand-written JavaScript counterparts when compiled to JavaScript [Ekblad, 2015a], we feel confident in judging Aplite's performance relative to Haste Haskell even so.

The results for Chrome are shown in figure 32, and the results for Firefox in figure 33. Execution times are given in milliseconds on a logarithmic

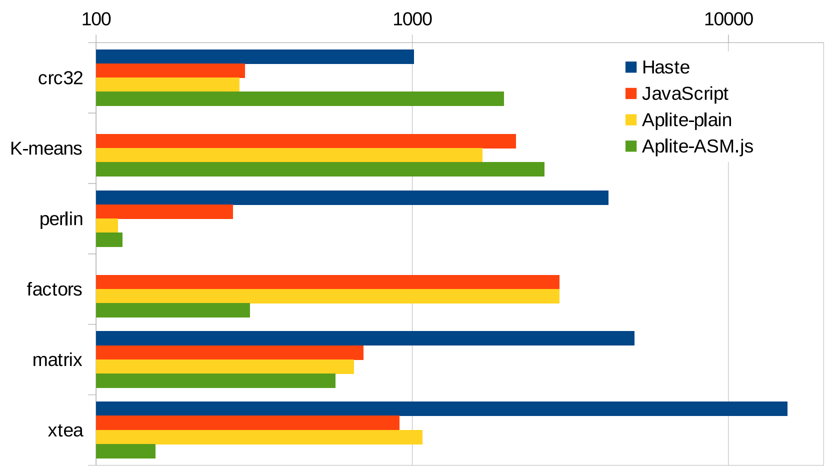

Figure 32: Chrome execution times in milliseconds

scale. The JavaScript bars indicate programs hand-written in JavaScript, while "Aplite-plain" and "Aplite-ASM.js" refer to Aplite's plain JavaScript and ASM.js backends respectively.

We can see that the additional array copying incurred by the ASM.js backend is taking its toll in the *crc32*. Modifying the *crc32* benchmark to avoid copying, by manually writing the input array into the ASM.js heap at compile-time, its execution time drops to slightly above 300 milliseconds – almost on par with the hand-written JavaScript or plain JavaScript backend – confirming our suspicion that the extra array copying is the culprit. The *K-means* and *matrix* benchmarks, which also deal with relatively large arrays, do not see much improvement from being altered similarly however, both being more computationally heavy than *crc32* and dealing with far smaller arrays. Regardless, performance of the ASM.js backend is clearly tilted in favour of computation-intensive workloads rather than memory-intensive ones.

The *factors* benchmark on Firefox provides one of the more disappointing results: both backends are just about equally performant as the hand-written JavaScript version. As this function is comparatively short and simple however, it stands to reason that browsers would be able to optimise both versions equally. This does not explain the results for the same benchmark on Chrome however, where the ASM.js backend produces a tenfold performance improvement. This goes to show that JavaScript performance can be quite unpredictable, even under the best of circumstances.

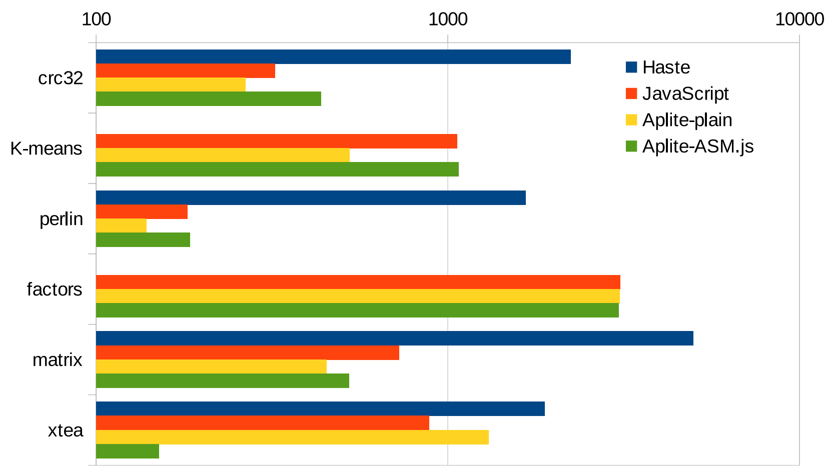

Figure 33: Firefox execution times in milliseconds

Meanwhile, the *xtea* benchmark delivers a speedup of nearly an order of magnitude on both browsers. Curiously, this is the only benchmark for which Firefox – the originator of ASM.js – produces better results with the ASM.js backend than with the plain JavaScript one. On Chromium, this picture is more nuanced, with the ASM.js backend giving the best results for half of the benchmarked programs.

Summarising the results, we see Aplite outperforming hand-written JavaScript in most benchmarks, sometimes by a large margin, and beating non-Aplite Haskell in every benchmark. While its performance relative to hand-written JavaScript varies by benchmark and browser, for each benchmark and browser there is always at least one Aplite backend that performs at least as well as the hand-written JavaScript implementation.

The increased run-time performance comes at a slight cost: Aplite programs need to be compiled before they can be executed, and as a multi-stage language, this compilation happens at run-time, in the user's browser. This cost is not excessive, however: all benchmarks presented in this section have compilation times from 20 to 180 milliseconds, depending on the benchmark, browser and backend. The ASM.js backend is generally faster than its plain JavaScript counterpart, and Firefox generally compiles Aplite programs faster than Chromium. Compilation times were measured as the difference between the first run of an Aplite function and a subsequent run over identical input, as the compilation process is mostly lazy. This means that the compilation times given here are slightly longer than the time actually spent in the Aplite code generator, but give a truer picture of what overhead to expect using Aplite.

6 Discussion and Related Work

Multi-Stage Programming A combined Haskell-Aplite program is essentially a special case of multi-stage programming, where the stages are restricted to Haskell compile-time, Haskell run-time (or Aplite compile-time), and Aplite run-time. Taha et al explored a more general system for multi-stage programming with their *MetaML* language [Taha and Sheard, 2000]. Although more expressive in its generality and offering intriguing possibilities for specialisation on multiple levels, their approach is somewhat less suitable for our purposes. Being embedded in Haskell, Aplite allows programmers to use the full expressive power of stock Haskell, including a vast collection of community-provided libraries. More crucially, Aplite also derives much of its speed from its *lack* of generality. Giving Aplite programs the ability to specialise and compile an arbitrary number of Aplite stages would add optimisation-unfriendly dynamism to all but the innermost stage. It is possible that clever use of multi-stage programming would enable performance gains that would offset the penalty thus incurred, but this still remains to be investigated. Rompf and Odersky present another interesting approach to multi-stage programming with *Lightweight Modular Staging* [Rompf and Odersky, 2010]. Like Aplite, but unlike MetaML, LMS is based on embedded DSLs as opposed to being built into the host language. Rather than a complete EDSL, LMS provides a framework for building multi-stage DSLs – an approach worth investigating if one were to extend Aplite with further stages.

If one were to investigate a more general multi-stage programming system for the web further, the *Sunroof* EDSL by Bracker and Gill [Bracker and Gill, 2014] demands further consideration. Like Aplite, Sunroof is a language embedded in Haskell which compiles down to JavaScript code. Unlike Aplite, Sunroof targets the full generality of the JavaScript language instead of a highly performant subset, going so far as to confine the Haskell parts of the system to the server. This approach does not lend itself well to performance-critical computations. Network latencies are both staggeringly high compared to the run-time of even the most expensive of the benchmarks in section 5 and highly unpredictable, and supporting the full JavaScript language places higher demands on programmer discipline to avoid straying into optimisation-unfriendly territory. A Sunroof implementation where the Haskell implementation could execute on the client would likely provide at least a modest performance improvement over plain Haskell, however. Such a system might play an important part in a more expressive multi-stage programming environment for web applications, adding an intermediate Sunroof stage between the Haskell and Aplite stages.

While not properly another stage in that it would not allow another layer of specialisation, such a system may be even further generalised through a multi-tier programming system such as *Haste.App* by Ekblad and Claessen [Ekblad and Claessen, 2014]. Haste.App allows a Haskell program to be automatically sliced into a server and a client part, giving a seamless and type-safe abstraction over the client-server communication that web applications must otherwise contend with. Integrating the aforementioned pieces would yield a combined web development environment encompassing server-side Haskell, client-side Haskell, Sunroof and Aplite.

The *Feldspar* EDSL by Axelsson et al [Axelsson et al., 2010], the ideas of which Aplite builds upon, might be used to extend such an environment even further: similar to how Aplite allows efficient computation on the client, integrating Feldspar via Haste.App would give the server added computational capabilities for the cases where even Aplite is not fast enough, or where the results of the computation are for some reason only relevant to the server.

While still not as general as MetaML, such an environment would allow programmers considerable flexibility in choosing the right tool for each part of their application, within the same language framework. One may discuss at which point adding more stages and layers to such a system becomes a curse rather than a blessing: going deeper may offer more power, but may also cause considerable confusion [Nolan, 2010]. Nonetheless, we feel that this is definitely a venue worth exploring.

High-Performance EDSLs Aplite is not the only EDSL to make loading of compiled code easy. Both Feldspar and the *meta-repa* language by Ankner and Svenningsson [Ankner and Svenningsson, 2013] use Template Haskell to automate the loading of their respective programs. Feldspar compiles down to highly performant C code which must then be compiled separately with some C compiler and manually loaded, making it on the surface relatively cumbersome to deal with. Using Template Haskell however, the entire process – from Feldspar to C to machine code being loaded into memory – may be automated [Persson, 2014]. Originally developed in order to facilitate testing, enabling Feldspar programs to be easily executed and examined within Haskell, the approach could also be used to speed up native Haskell programs using Feldspar, similar to what Aplite does for web-targeting Haskell programs. As this process happens entirely during compile-time, it does not allow Feldspar programs to be specialised over their execution environment or user input, however. Meta-repa takes another approach. Instead of compiling down to C, it uses Template Haskell to compile down to highly specialised and efficient Haskell code instead. Unlike Feldspar, this avoids the dependency on an external C compiler, allowing the whole compilation process to take place within the Haskell compiler. As with Feldspar, this process happens entirely at compile-time and programs can thus not be specialised to run-time factors. Although many EDSLs opt to execute their entire compilation pipeline at compile-time, tapping into C or Haskell compilers that may not be available in their target environment, projects like *Harpy*, a machine code generation DSL by Grabmüller and Kleeblatt, demonstrate that generating and loading native code at run-time is most definitely a viable option [Grabmüller and Kleeblatt, 2007].

Taking high-performance EDSLs even further, several such languages target GPUs rather than conventional CPUs [Mainland and Morrisett, 2010, McDonell et al., 2015, Svensson et al., 2010]. One such language, *Nikola* by Mainland and Morrisett, is quite similar to Aplite: both languages provide on-demand, run-time compilation (and, consequently, the ability to specialise over input and environment) of n-ary functions and seamless integration between host and embedded code. In addition, Nikola supports the verbatim inclusion of code written in its target language; something which Aplite does not. Nikola functions are always pure, due to the strict separation between the CPU executing the host program and the GPU executing the Nikola kernels. In contrast, Aplite kernels may perform some side-effects visible to the host program and vice versa, giving the impetus for Aplite's use of type families to determine which functions can be imported purely and which can not. Aplite also improves upon Nikola by introducing multiple backends and allowing automatic specialisation of kernels to the each kernel's input and execution environment.

Efficient JavaScript Compilation There are also more all-or-nothing approaches to the issue of producing efficient JavaScript. The ASM.js JavaScript subset was first conceived as a compilation target for general purpose languages, more specifically Mozilla's *Emscripten* compiler [Zakai, 2011]. Through its LLVM frontend, Emscripten is able to compile a wide range of languages down to efficient JavaScript. There are, however, reasons why one would rather use a language which directly targets JavaScript in conjunction with a performance-oriented EDSL. For one, web browsers ship with extensive run-time functionality, which programs compiled with Emscripten need to re-implement on their own. This may partially negate some of its efficiency gains, but more crucially it involves shipping larger code to users. While the size of a native binary stored on a hard drive is usually not an issue, the prospect of sending tens of megabytes of JavaScript to every user before their application can even begin to load certainly is. Additionally, not all browsers support ASM.js and, as we have seen, not all programs see performance benefits from using it.

There is an initiative – WebAssembly – to replace ASM.js with a cross-browser compatible, binary format with the same semantics [Eich, 2015]. While this may somewhat mitigate the large code sizes that affect ASM.js programs, it is still not a perfect fit for high-level languages, much for the same reasons that ASM.js is not. Work is ongoing on improving this standard however, so it is not inconceivable that it would one day make a compelling alternative to compiling high-level languages directly to JavaScript. However, even should that day come, high-performance EDSLs still have their place. High-level languages for native platforms are usually outperformed by highly optimised domain-specific (or just lower-level) languages.

6.1 Limitations

Mitigating Copying Costs As we showed in section 5, using Aplite brings major performance benefits over plain Haskell code and as well as over hand-written JavaScript. This does, however, assume that the correct back-end for the task at hand is chosen. The benchmarks show clearly that using the wrong backend may impact performance severely, part of which is caused by the extra copying of arrays incurred by the ASM.js backend.

This performance penalty might be mitigated in a number of ways. The lowest-hanging fruit in this area would be introducing immutable arrays. In the benchmark where extraneous copying hurt the most – *crc32* – the input array is never mutated. Recall that half the cost of array copying is incurred when a function returns, when data is copied out from the heap into an array in order to make mutation visible to the outside world. For an immutable array we would not incur this cost at all, as it can obviously not have been mutated, which means that its representations in the original array and in the ASM.js heap must be identical, and so there is no need to copy the array's contents back out again.

Too Many Variables When loading JavaScript code dynamically as described in section 3, certain web browsers – including both Chrome and Firefox – refuse to optimise functions over a certain size, as measured by the number of local variables. None of our benchmarks have hit this limitation, but as this limit is relatively low – between 300 and 600 variables – it is quite conceivable that large Aplite functions would hit upon this limit. As Aplite relies heavily on metaprogramming, inlining functions rather than calling them, this may present a problem. As the local variable limit is counted per-function, automatically breaking Aplite programs into appropriately sized chunks, placing each in a separate function, would solve this problem neatly although possibly at the expense of a slight performance hit.

High-Level Abstractions Apart from the powerful metaprogramming capabilities afforded by the use of Haskell as its host language, Aplite as presented is sporting a relative lack of high-level abstractions for a functional language. Efficiently implementing high-level abstractions over arrays and numbers is not impossible; the rich set of abstractions present in the Feldspar language is ample proof of this. As described in section 2, Aplite is designed to act as a fast, low-level core for higher-level abstractions. As such, extending the language's user interface with more convenient constructs is relatively easy. This is partially employed, albeit in a more ad hoc manner, in the benchmarks, where Haskell-level metaprogramming is used to implement, among other things, virtual data structures and unrolled bounded recursion over lists.

7 Conclusions and Future Work

Conclusions We have presented the Aplite embedded domain-specific language for high-performance, client-side web applications. Its multi-staged programming model allows programmers to seamlessly mix high-level control flow code written in Haskell with highly efficient, low-level computational kernels written in Aplite. Aplite builds on previous work on high-performance EDSLs, adapting the concepts to the web domain where unpredictable performance of idiomatic JavaScript makes a strong case for compiling some higher-level metalanguage down to a predictably efficient JavaScript subset. Its capability to provide predictable performance is further enhanced by the use of multiple backends, with automatic run-time backend selection based on the measured performance of Aplite programs, allowing the same source code to be compiled in the most suitable manner possible for any given task. The use of Haskell as a metalanguage enables the practical use of partial evaluation and multi-stage programming, giving developers fine-grained tools to optimise their computational kernels by specialising them to user input, and experimenting with different optimisations at their leisure.

Depending on the task, Aplite can bring performance benefits of an order of magnitude over both non-Aplite Haskell code and hand-written

JavaScript. While not all of its backends perform equally well for all programs and browsers, for each investigated program and browser there is at least one Aplite backend which performs at least as well as hand-written, idiomatic JavaScript, and built-in support to automatically select the proper backend for each situation. We claim that performance-oriented EDSLs are eminently applicable to the domain of web applications. In implementing Aplite, we have believe we present strong evidence that this is indeed the case.

Future Work While we have shown that Aplite fulfils its promise of enabling high-performance functional web applications, we are just beginning to explore the design space. Certain programs in the space targeted by Aplite may benefit from execution on a GPU. Given that Aplite's intermediate representation is already severely restricted, exploring the possibilities of a GPU backend may be a good way forward. Similarly, the capability to run programs in parallel on different CPU cores is another possibility worth exploring. Such backends – or perhaps even a dedicated GPU stage – may make useful building blocks in the more encompassing multi-stage web programming environment we envisioned in section 6. Another interesting possibility would be a *hybrid backend*, in which parts of the code is compiled into ASM.js and parts into plain JavaScript. This offers the possibility of taking advantage of ASM.js for computation-intensive parts of the program, while mitigating or avoiding the copying penalty for memory-intensive parts.

Presently, communication between Haskell and Aplite is one-way: Haskell calls, and Aplite responds. As ASM.js has some limited FFI capabilities, it would be possible to give Aplite programs the capability to call back into Haskell, provided that the called functions have types that are compatible with the rather limited set of types approved for the ASM.js FFI. This would add interesting prototyping capabilities: Aplite programs could be written in plain Haskell at first, and gradually converted into pure Aplite as the design of the application solidifies or the performance demands grow. This also ties into the issue of higher-level abstractions discussed in section 6.1. While our language presently shows eminent performance, implementing higher-level abstractions remains a high-priority future work item.

Acknowledgements

Many thanks to Koen Claessen, Emil Axelsson, Michał Pałka and Atze van der Ploeg for their valuable feedback, discussion and encouragement.

This work was funded by the Swedish Foundation for Strategic Research, under grant RAWFP.

8 Bibliography

J. Ankner and J. D. Svenningsson. An EDSL approach to high performance Haskell programming. In *ACM SIGPLAN Notices*, volume 48, pages 1–12. ACM, 2013.

P. Antonov. Optimization killers. https://github.com/petkaantonov/bluebird/wiki/Optimization-killers, 2015.

E. Axelsson. The imperative-edsl package. http://hackage.haskell.org/package/imperative-edsl, 2015.

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.

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. ISBN 978-3-319-04131-5. doi: 10.1007/978-3-319-04132-2_5. URL http://dx.doi.org/10.1007/978-3-319-04132-2_5.

E. Czaplicki. Elm: Concurrent FRP for functional GUIs. *Senior thesis, Harvard University*, 2012.

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. ISBN 978-3-642-41581-4. doi: 10.1007/978-3-642-41582-1_3. URL http://dx.doi.org/10.1007/978-3-642-41582-1_3.

B. Eich. From ASM.js to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/, 2015.

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. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535856. URL http://doi.acm.org/10.1145/2535838.2535856.

A. Ekblad. *A Distributed Haskell for the Modern Web*. Licentiate thesis, Chalmers Institute of Technology, 2015a.

A. Ekblad. Foreign exchange at low, low rates. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, pages 2:1–2:13, New York, NY, USA, 2015b. ACM. ISBN 978-1-4503-4273-5. doi: 10.1145/2897336.2897338. URL http://doi.acm.org/10.1145/2897336.2897338.

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. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633367. URL http://doi.acm.org/10.1145/2633357.2633367.

P. Freeman. PureScript. http://www.purescript.org/, 2015.

Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

M. Grabmüller and D. Kleeblatt. Harpy: Run-time code generation in Haskell. In *Haskell Workshop 2007*. ACM Press, September 2007.

D. Herman, L. Wagner, and A. Zakai. The ASM.js draft specification. http://asmjs.org/spec/latest/, 2014.

G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. *ACM Sigplan Notices*, 45(11):67–78, 2010.

T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton. Type-safe runtime code generation: Accelerate to LLVM. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 201–212, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3808-0. doi: 10.1145/2804302.2804313. URL http://doi.acm.org/10.1145/2804302.2804313.

V. Nazarov, H. Mackenzie, and L. Stegeman. GHCJS Haskell to JavaScript compiler. https://github.com/ghcjs/ghcjs, 2015.

C. Nolan. Inception. http://www.imdb.com/title/tt1375666/, 2010.

K. Perlin. Improving noise. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 681–682. ACM, 2002.

A. Persson. Towards a functional programming language for baseband signal processing. 2014.

T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *ACM SIGPLAN Notices*, volume 46, pages 127–136. ACM, 2010.

SheetJS. Sheetjs. http://sheetjs.com/, 2014.

J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming*, pages 21–36. Springer, 2012.

J. D. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *ACM SIGPLAN Notices*, volume 48, pages 299–304. ACM, 2013.

J. Svensson, K. Claessen, and M. Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2065–2074, 2010.

W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1):211–242, 2000.

J. Vouillon and V. Balat. From bytecode to JavaScript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.

A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

A. Zakai. Big web app? Compile it! http://kripken.github.io/mloc_emscripten_talk/, 2013.

Paper VI

Scoping Monadic Relational Database Queries

Abstract

We present a novel method for ensuring that relational database queries in monadic embedded languages are well-scoped, even in the presence of arbitrarily nested joins and aggregates. Demonstrating our method, we present a simplified version of *Selda*, a monadic relational database query language embedded in Haskell, with full support for nested inner queries. To our knowledge, Selda is the first relational database query language to support fully general inner queries in a monadic interface.

In the Haskell community, monads are the de facto standard interface to a wide range of libraries and EDSLs. They are well understood by researchers and practitioners alike, and they enjoy first class support by the standard libraries. Due to the difficulty of ensuring that inner queries are well-scoped, database interfaces in Haskell have previously either been forced to forego the benefits of monadic interfaces, or have had to do without the generality afforded by inner queries.

1 Introduction

One of the most common tasks in software development is querying databases for information, and SQL-based relational databases in particular. Most languages provide some sort of support for this task, either as standard library functionality or as built-in language primitives. Regardless of its exact implementation, this support often takes the form of an *embedded language*: a high-level interface in which programmers can describe queries directly in the host language, instead of writing raw SQL queries as strings to be passed to the database engine. The embedded language approach has many advantages for database integration: queries are correct by construction, can be type-checked together with the host program, and often allows one to write queries at a higher level of abstraction.

In Haskell, *monads* [Wadler, 1995] are the de facto standard interface for embedded languages. They are widely studied by researchers, and thanks to their ubiquity every semi-proficient Haskell programmer is familiar with them. The standard Haskell libraries provide a wide range of functions over monads, and the language itself includes syntactic sugar to make monadic code more readable and concise.

1.1 Monads for embedded languages

For the aforementioned reasons, having a monadic interface is often desirable for embedded languages when possible. Using another abstraction often incurs an unnecessary cost in time and effort for new users adopt

```
addresses = do
  (name :*: addr) ← select persons
  city ← leftJoin (\city → addr .== city) $ do
    (city :*: country) ← select cities
    restrict (country .== "Sweden")
    return city
  return (name :*: city)
```

Figure 7.1: Obtaining the Swedish address of each person in a database

the library or embedded language. Any given Haskell programmer is very likely to already be familiar with the monad abstraction, making the adoption of a monadic embedded language easier. The same programmer is significantly less likely to be familiar with other abstractions. When a general abstraction is unfamiliar, its use may instead make the embedded language *harder* to adopt, since the programmer also needs to learn new abstractions not directly related to the domain of the problem they are trying to solve.

There is precedent for using a monadic interface to query collections in Haskell. The built-in list type makes up a monad, where the bind operation is equivalent to the Cartesian product of two lists. As discussed in Sect. 4.1, Haskell also has several monadic embedded languages for querying databases, all of which are lacking in generality compared to equivalent non-monadic languages, however.

1.2 Scoping monadic queries

The lack of expressiveness in other monadic languages comes from a difficulty of scoping certain queries. Consider the query in Fig. 7.1, which selects each person from a table *persons* and associates each person with their home city, but only if that city is located in Sweden. The *inner query* making up the second argument of the *leftJoin* function is joined to the *current result set*, or the Cartesian product of the queries performed so far. It is equivalent to the SQL query in Fig. 7.2.

While the program from Fig. 7.1 is not problematic in itself, the fact that the *name* and *addr* identifiers, drawn from the *persons* table, are in scope *inside the body of the join* could enable us to write queries that are not well-scoped. Fig. 7.3 shows how this query might be written in an ill-scoped manner. The result of the right side of an SQL join operation is not in scope on the left side and vice versa. Thus, the inclusion of an identifier from the scope of the current result set, which makes up the left

```
SELECT personName, cityName
  FROM persons
  LEFT JOIN (
    SELECT cityName
    FROM cities
    WHERE cities.country = "Sweden"
  )
  ON persons.address = cityName
```

Figure 7.2: Obtaining the Swedish address of each person, in SQL

```
illScopedAddresses = do
  (name :*: addr) ← select persons
  city ← leftJoin (\city → addr .== city) $ do
    (city :*: country) ← select cities
    restrict (country .== "Sweden")
    restrict (city .== addr)
    return city
  return (name :*: city)
```

Figure 7.3: An ill-scoped join

side of the join, into the body of the right side is a violation of SQL scoping rules.

The same problem arises for aggregate queries, as well as any other type of join operation. To support these operations in a monadic embedding of relational database queries, we need a way to disallow ill-scoped queries like that in Fig. 7.3, while still allowing queries like the one in 7.1.

Contributions With this paper, we extend the expressiveness of monadic database interfaces to cover an arbitrary nesting of joins and aggregate queries. Our contribution consists of (1) a method for ensuring that relational database queries are well-scoped in the presence of inner queries, and (2) *Selda*, a monadic language for database queries embedded in Haskell, demonstrating the use of the aforementioned method.

2 A basic query language

Before explaining our method of scoping monadic query languages from Sect. 3 on, we must first briefly describe the query language we are going to scope. The language presented in this paper, dubbed *Selda* as an embarrassing pun [Miyamoto, 1986] on *LINQ* [Meijer et al., 2006], uses the semantics of the list monad, with the bind operation denoting the Cartesian product of its elements, as its starting point. From there, we extend this basic programming model with projection, restriction, aggregation, join operations, and many other features. The version of Selda presented in this paper is significantly simplified, to focus on the issue of scoping inner queries. The full version is freely available from the project website.[1]

2.1 The Selda programming model

Selda Queries are written in the *Query* monad, which is parameterised over an extra type *s*. This type parameter is discussed at length in Sect. 3 but for now, we will ignore it. Queries are performed over *column expressions* of type *Col*, which also shares this type parameter. As in the list monad, the bind operation denotes the Cartesian product of its two arguments. We call this Cartesian product the *current result set*.

Queries are compiled into SQL and executed by a relational database engine using the *query* function. The *Row* class and its associated type *FromRow* is responsible for converting the results of a query back to a Haskell-level list of values. The *Columns* type class denotes any type that is a heterogeneous list of columns, and the *Aggregates* class implements the same restriction for *aggregated* columns, which are discussed in Sect. 3.2. The latter two classes are necessary to ensure that only values from the Selda language itself are propagated through queries.

Note that all three classes are closed, and have no user-accessible methods. If one were to open them up for extension, it would be possible to create an instance of *Columns* or *Aggregates* that subverts the scoping restrictions described in Sect. 3, making the language as a whole ill-scoped.

For reference, the API of our simplified language is given in Fig. 7.4.

2.2 Result rows as heterogeneous lists

Result rows in Selda are represented as *heterogeneous lists* of columns, much like the *HLIST* data structure by Kiselyov et al. [2004]. This allows us to define types and functions inductively over result rows, avoiding the metaprogramming used by many embedded languages to define operations over database results. As we assume all query results to contain at least one column, we define our heterogeneous lists to be non-empty. A heterogeneous list is defined as one or more values interspersed with the *:*:* operator. Note that this means that a plain value of some type is *also* a "list" of a single element. Examples of such lists would be *(1 :*: 2)*, *"foo"*, and *(42 :*: "I'm a little teapot" :*: ())*. The *:*:* operator is defined as follows.

[1]https://selda.link

```haskell
-- Types and classes
data Table a
data Query s a
data Col s a
data Aggr s a

data Inner s
type family Outer a

class Columns a
class Aggregates a
class Row a where
  type FromRow a
instance Monad (Query s)

-- Executing queries and defining tables
query :: Row a ⇒ Database → Query s a → IO [FromRow a]
table :: Columns a ⇒ Text → a → Table a

-- Query operations
select :: Table a → Query s a
restrict :: Col s Bool → Query s ()
inner :: Columns a ⇒ Query (Inner s) a → Query s (Outer a)
aggregate :: Aggregates a ⇒ Query (Inner s) a → Query s (Outer a)
leftJoin :: Columns a
         ⇒ (Outer a → Col s Bool)
         → Query (Inner s) a
         → Query s (Outer a)
...

-- Expressions
instance Num a ⇒ Num (Col s a)
(.>), (.<), ... :: Ord a ⇒ Col s a → Col s a → Col s Bool
min_, max_, ... :: Ord a ⇒ Col s a → Aggr s a
count :: Col s a → Aggr s Int
...
```

Figure 7.4: API of our simplified language

```
persons :: Table (Col s Text :*: Col s Int :*: Col s Text)
persons = table "persons" $ column "name"
                        :*: column "age"
                        :*: column "city"

getMinorStatus :: Query s (Col s Text :*: Col s Bool)
getMinorStatus = do
  (name :*: age :*: _) ← select persons
  return (name :*: age .< 18)
```

Figure 7.5: Table definition, projection and expression in Selda

```
data a :*: b where
  (:*:) :: a → b → a :*: b
infixr 1 :*:
```

In Selda, heterogeneous lists are used consistently as a replacement for tuples. Queries receive and return heterogeneous lists, and the *FromRow* type family used by the *query* runner function maps heterogeneous lists over columns to heterogeneous lists over their corresponding Haskell-level types. More formally, *FromRows (Col s a1 :*: ... :*: Col s an)* ≡ *(a1 :*: ... :*: an)*.

2.3 Tables, expressions and basic queries

At the base of every query lies one or more database tables. Selda provides a simple table definition language, where the Haskell types and SQL names of the table and its columns are given to create a *Table* value. The *Table* type is parameterised over a heterogeneous list of column types.

Tables are queried using the *select* function, which return a heterogeneous list of tuples corresponding to the table's column types, which may then be pattern matched by the querying computation. Queries may perform *projection* by simply returning the values they want to project.

Columns have the type *Col s a*, and may represent either a column from an actual table, or an *expression* over zero or more such columns. Column expressions may be constructed using the expected operations: addition, subtraction, comparisons, boolean operations, etc. It is important to note that *Col* is an abstract type; a requirement for being able to reify the structure of a monadic computation over columns [Svenningsson and Svensson, 2013].

Fig. 7.5 demonstrates how to define a table *persons* and query it to associate each person with a boolean column expression denoting whether the person is a minor.

```
getMinors :: Query s (Col s Text)
getMinors = do
  (name :*: age :*: _) ← select persons
  restrict (age .< 18)
  return name
```

Figure 7.6: Table definition, projection and expression in Selda

Queries may be filtered using the *restrict* operation. If a query computation contains a call to *restrict p*, only result rows for which *p* evaluates to true will be kept in the current result set. This is akin to the filtering operation of Haskell's list comprehensions, or the standard library function *filter*. Fig. 7.6 modifies the query from Fig. 7.5 to *remove* all non-minors from the result set, instead of merely tagging each person with their minor status.

3 Inner queries

Having dispensed with the preliminaries, we now turn to the main contribution of this paper. As we discussed in Sect. 1.2, we want to support general inner queries in aggregates and join operations, while guaranteeing that column expressions from an outer query do not leak into an inner one. To accomplish this, we leverage Haskell's expressive type system. Namely, *phantom types* [Cheney and Hinze, 2003] – purely nominal type parameters – and *closed type families* [Eisenberg et al., 2014] – the Haskell flavor of functions at the type level rather than at the value level.

3.1 If all else fails, use type variables

Recall that the *Query* and *Col* types discussed in Sect. 2.1 are parameterised over an extra type variable *s*. The reason for this extra parameter, as the attentive reader might have started to suspect by now, is to keep track of the scope of a column expression, ensuring that each database operation only ever interacts with column expressions in "their own" scope.

Much like Haskell's *ST* monad, all Selda operations over columns require that the scope type variable of the *column* matches the scope variable of the *operation*. This ensures that inner queries can not touch columns from outer queries and vice versa, as long as we manage to ensure that the inner and outer queries can never share the same scope type variable.

To this end, we introduce a type *Inner* to our language to distinguish different scopes, as well as a type family *Outer*, which hoists the type of a

```haskell
data Inner a

type family Outer a where
  Outer (t (Inner s) a :*: b) = Col s a :*: Outer b
  Outer (t (Inner s) a)       = Col s a
```

Figure 7.7: Scope hoisting for inner queries

heterogeneous list of results in scope *Inner s*, to the equivalent heterogeneous list type of results in scope *s*, as shown in Fig. 7.7.

Then, any inner query *q′* of some query *q :: Query s (Outer a)* must have the type *Query (Inner s) a*. We enforce this by ensuring that any function which takes an inner query as an argument forces the *s* parameter of the query to denote a scope *one level deeper* than the current scope. See for instance the types of *inner* and *leftJoin*:

```haskell
inner :: Columns a ⇒ Query (Inner s) a → Query s (Outer a)


leftJoin :: Columns a
         ⇒ (Outer a → Col s Bool)
         → Query (Inner s) a
         → Query s (Outer a)
```

As all column expressions are parameterised over the scope in which they originated, and as all operations over columns force the scope of the columns to the present scope, this prevents columns from an outer scope from being used in an inner query. Columns from an inner scope are prevented from escaping to an outer one by the *Columns* constraint on the return values of inner queries – that they are always heterogeneous lists of columns. This restriction, together with the "scope hoisting" enforced by the *Outer* type family ensures that scopes are never mixed. Sect. 4 discusses this property in more detail.

3.2 Aggregation

Aggregate queries give rise to the same problems with scoping as other inner queries: as aggregation "collapses" its input query, it can not share its input with any non-aggregate query, or even any other aggregation. Fortunately, using the phantom type technique described above solves this problem as well, as long as we give our aggregation function an appropriate type:

```
aggregate :: Aggregates a
          ⇒ Query (Inner s) a
          → Query s (Outer a)
```

Note that the type class constraint of this function differs from the one on, for instance, the *inner* function. In addition to the more general scoping problem, aggregated queries come with a related problem of their own. Consider the following query.

```
allAdults :: Query s (Col s Text)
allAdults = aggregate $ do
  (name :*: age :*: city) ← select persons
  restrict (age .> min_ age)
  return name
```

The equivalent SQL query would be:

```
SELECT name
  FROM persons
  WHERE age > MIN(age)
```

While this query may intuitively make sense – return the names of every person who is older than the youngest person – this is another violation of SQL's scoping rules: `WHERE` clauses must not refer to aggregate expressions. Taking a closer look at the query, we see that there is a good reason for this. Aggregate expressions compute their value over the aggregate of the whole query *after restrictions are applied*. Thus, allowing restrictions to refer to aggregates over the current query would lead to an infinite loop.

At first glance, we might be tempted to solve this problem by scoping all aggregate expressions one level deeper than the current scope:

```
min_ :: Ord a ⇒ Col s a → Col (Inner s) a
```

However, this solution breaks down in the presence of nested inner queries. If a query of scope *s* is able to create column expressions of scope *Inner s*, those column expressions could then be accessed from within any inner query in the same scope. This would effectively make the scoping control we've introduced so far useless!

Instead, we opt to use a different column type entirely for aggregated columns, which we call *Aggr*, and assign appropriate types to our aggregation functions:

```
min_, max_, ... :: Ord a ⇒ Col s a → Aggr s a
count :: Col s a → Aggr s Int
...
```

The attentive reader may have noticed that the definition of the *Outer* type family in Fig. 7.7 is unnecessarily general: why convert *t (Inner s) a* into *Col s a* instead of the more obvious choice of converting *Col (Inner s) a* into *Col s a*? The *Aggr* type provides the answer: *Outer* does not only hoist "plain" column expressions into the outer scope, but aggregated expressions as well.

It should be noted that only aggregate column expressions may be returned from an aggregated inner query, as enforced by the *Aggregates* type class constraint. While some SQL dialects allow non-aggregated expressions to be projected from an aggregate query, the semantics of this are unclear at best.

4 Discussion and related work

As discussed in Sect. 1.1, there are several advantages to using a monadic interface when embedding a language into Haskell: good library support, syntactic sugar, and programmer familiarity. However, the disadvantage is relatively obvious: not all languages are equally suited to a monadic interface, and even the languages that are, may require a less obvious implementation than when using another abstraction. As the current state of the art shows, this can be said to be the case for database query languages. However, while the method presented in this paper may be slightly trickier to implement than some other relational database abstraction, it does not complicate the language for the end user. Instead, it enables our language to reap all of the aforementioned advantages of monadic interfaces.

Relation to the *ST* monad Haskell-savvy readers may have picked up on the fact that our method is strikingly similar to the method introduced by Launchbury and Peyton Jones [1994] to allow computations with a referentially transparent interface to safely use mutable references internally. This is accomplished by introducing a monad *ST s a*, in which references of type *STRef s a* may be created and mutated. Like with our method, functions in the monad are parameterised over a state thread *s* (e.g. *newSTRef :: ST s (STRef s a)*, to ensure that computations can only ever interact with references created in the same state thread. Computations in the monad are executed using a function *runST :: (forall s. ST s a)* $\rightarrow$ *a*, which ensures that each stateful computation is executed in its own state thread. Just like with inner queries, with state threads we have multiple separate computations in the same monad, which must not be allowed to freely interact with one another.

Unfortunately, this method does not work for inner queries. While the ST monad is intended to stop any *and all interaction* between state threads, our language needs allow interaction between queries *in a controlled manner*. As we saw in Sect. 3, the body of an inner query returns a list of columns, where each column has the type *Col (Inner s) a*. This list is then converted into a list of columns of type *Col s a* by the function (i.e. *inner*, etc.) used to execute it. Using the method by Launchbury and Peyton Jones, the *inner* function would instead have had the type *(forall s. Query s a)* $\rightarrow$ *Query s′ a*. Looking at this type, we can see that it will not allow us to return any type *a* which references *s*. The *s* type variable is bound by the *forall* quantifier of the *Query s a* type. Referencing *s* in *a* would allow it to escape its scope, as *a* occurs outside the quantifier's scope.

Note that, just like this method is not applicable to our problem, our method is not applicable to state threads. Our method does not make use of existential quantification to ensure the uniqueness of each state thread, but instead relies on the top level query to fix the "base" *s*. This means that any application which must be callable from referentially transparent code would either be unsafe, as there would be no way to guarantee the global uniqueness of state threads, or be forced to use an existentially quantified "base" *s* anyway, which would make the rest of our method no more than unnecessary line noise.

Relation to type-level natural numbers Our method essentially encodes the nesting level of the current scope as a type-level natural number, with the *s* determined by the *query* runner function as the zero value, and each consecutive application of the *Inner* type constructor acting as an application of the successor function. This may sound worrisome at first: it is definitely the case that two separate nested queries may have the same nesting level. How, then, can we be sure that their scopes don't leak into each other? The answer lies in the return type of the inner query functions. By only allowing lists of columns to be returned, decrementing the nesting level of each column by one, we ensure that no column is able to escape without having its nesting level properly decremented. Note that this property relies on the inability of the query expression language to nest columns within columns. If a column of type *Col s (Col s′ a)* could be constructed and inspected, further measures would be required to ensure that the nesting level of the inner column would not escape its scope.

4.1 Related work

The area of relational database EDSLs is an active one, in academia as well as in industry. Arguably, the most well-known such language is *LINQ* [Meijer et al., 2006]. Embedded in the .NET framework, LINQ allows SQL-like queries to be made over not only databases, but any suitable collection. LINQ uses a more restricted and SQL-like, streaming interface which avoids the problem of scoping inner queries entirely, by only allowing inner queries in contexts where no identifiers from the outer scope are bound. *ScalaQL* [Spiewak and Zhao, 2009] provides a similar interface for Scala, while the *Opaleye* EDSL [Ellis, 2014] uses arrows and profunctors to provide the same for Haskell. Similarly, the *Esqueleto* Haskell EDSL [Lessa, 2012] provides a continuation-based interface to database queries. Silva and Visser [2006] describe an ingenious ad hoc embedding of database queries into Haskell, capable of supporting functional dependencies and other advanced features. Augustsson and Ågren [2016] provide perhaps the most debuggable database EDSL to date, using user-defined type errors to great effect in their Haskell encoding of relational algebra.

All of the aforementioned approaches support fully general inner queries, but do so at the price of forgoing a monadic interface. There exists several monadic database EDSLs however. *HaskellDB*, perhaps the grandfather of database EDSLs, provides a monadic interface to database queries, but does not support inner queries due to the difficulty of typing them [Leijen and Meijer, 1999]. *Beam* [Athougies, 2016] and *Haskell Relational Record* [Hibino et al., 2015] are more recent attacks on the problem, which provide monadic interfaces that do support joins and aggregation, but only over database tables, and not over inner queries.

While Selda does not strictly enable any queries to be written that were not possible before, it does expand the set of queries that can be written using a monadic interface over the current state of the art.

5 Conclusions and future work

We have presented a monadic interface to relational database queries. We improve upon the state of the art by leveraging the host language's type system to ensure that even fully generic inner queries are well-scoped. To our knowledge, ours is the first monadic relational database EDSL to accomplish this. While our method fulfils its original design goal – enabling a monadic database EDSL in the spirit of the list monad – applying our method to other problems remains as future work. One possible such application would be in the area of cache-aware computations, where the hierarchical nature of our scoping method could provide static guarantees regarding prefetching and data access in a hierarchy of caches.

Acknowledgements

Many thanks to Mauro Jaskelioff, Tom Ellis, Koen Claessen, and Michał Pałka, for their valuable insights, feedback and discussion.

6 Bibliography

T. Athougies. Beam. http://travis.athougies.net/projects/beam.html, 2016.

L. Augustsson and M. Ågren. Experience Report: Types for a Relational Algebra Library. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 127–132, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4434-0. doi: 10.1145/2976002.2976016.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

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*. ACM, 2014. ISBN 978-1-4503-2544-8.

T. Ellis. Opaleye. https://github.com/tomjaguarpaw/haskell-opaleye, 2014.

K. Hibino, S. Murayama, S. Yasutake, S. Kuroda, and K. Yamamoto. Haskell relational record. http://khibino.github.io/haskell-relational-record/, 2015.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: 10.1145/1017472.1017488.

J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178246.

D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2Nd Conference on Domain-specific Languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM. ISBN 1-58113-255-7. doi: 10.1145/331960.331977.

F. Lessa. Esqueleto. http://hackage.haskell.org/package/esqueleto, 2012.

E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552.

S. Miyamoto. The Legend of Zelda.
https://en.wikipedia.org/wiki/Link_(The_Legend_of_Zelda), 1986.

A. Silva and J. Visser. Strong types for relational databases. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 25–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-489-8. doi: 10.1145/1159842.1159846.

D. Spiewak and T. Zhao. Scalaql: language-integrated database queries for scala. In *International Conference on Software Language Engineering*, pages 154–163. Springer, 2009.

J. D. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 299–304, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500611.

P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.