Foreign Exchange at Low, Low Rates
A lightweight, platform-agnostic FFI for high level Haskell

Anton Ekblad
Chalmers University of Technology
antinek@chalmers.se

Abstract

We present a novel yet simple foreign function interface for Haskell dialects targeting high level platforms. The interface automates marshalling, eliminates boilerplate code, allows increased sanity checking of external data, and is implementable as a plain Haskell ’98 library across a range of Haskell compilers and host environments without any modification to the 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 optimized with minimal effort to perform on par with Haskell’s vanilla foreign function interface, as well as extended to support automatic marshalling of functions and modified to support a larger range of host environments at the cost of a slight increase in implementation complexity.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (functional) Programming; D.2.11 [Software Architectures]: Languages; D.2.12 [Interoperability]: Interface definition languages

Keywords compilers; interoperability; web

1. Introduction

Interfacing with other languages is one of the more painful aspects of modern day Haskell development. Consider figure 1, taken from the standard libraries of GHC, a piece of code to retrieve the current time [21]. 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. But is this really the best we can do? During our work on the web-targeting Haste compiler [12], we have come to believe that, for high level target environments, it is not.

While Haste initially made use of the conventional Foreign Function Interface extension [4] to interface with its browser target environment, this presented certain difficulties. The modern web browser environment is highly reliant on callback functions and complex data types, none of which are trivial to pass through the FFI, making browser-interfacing Haste code relatively clunky and byzantine.

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

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

---

Copyright notice will appear here once 'preprint' option is removed.
specifically, we would like a foreign function interface for this level an interface for the domain of hosted Haskell dialects. More With this background, we believe that the vanilla FFI is too low level representations of the vanilla foreign function interface an same data structures and representations, making the forced low runtime with the host language and internally uses many of the language: the guest language commonly shares a large part of its for interfacing with code written in the respective compiler's host – – – – draft – – – –

Figure 2: Foreign imports using our FFI

ous higher level data structures, as there simply are no higher level data structures.

With the rise of the web browser as a major application platform, however, a whole new breed of hosted dialects of Haskell and other functional programming languages have emerged, compiling not only to JavaScript and the web browser platform, but to other high level languages as well. New languages like Idris [3] offer a multitude of backends, some of which are hosted, and compilers like Haste [12] and Ocsigen [2] take advantage of multiple backends to implement program slicing, producing programs which run concurrently distributed over a hosted context and an unhosted, down-to-the-metal one.

For these hosted language implementations, 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 code written in the respective compiler’s host language: the guest language commonly shares a large part of its runtime with the host language and internally uses many of the same 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 the vanilla FFI is too low level an interface for the domain of hosted Haskell dialects. More specifically, we would like a foreign function interface for this domain to espouse the following qualities:

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

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 2, without having to make intrusive changes to our Haskell compiler.

Contrasting this with the standard FFI code from figure 1:

- The low level C types are gone, replaced by a more descriptive record type, and so is the peeking and poking 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 FFI example in figure 1 had to import the gettimeofday system call by name, its actual implementation given elsewhere, we have actually implemented its JavaScript counterpart at the location of its import, without having to resort to external stubs.

As all languages have some manner of interfacing with foreign code, there already exists a large body of work in this problem domain. However, as we discuss in section 6.5, current solutions all fall short of the criteria presented here in some way.

A note on “hosted” Haskell Every programming language is in some sense hosted, where “hosting” refers to the act of providing said language with an environment in which to execute: assembly code is hosted by a physical machine and (usually) an operating system; some languages are hosted by C, using it as a compilation target and sharing its runtime system rather than compiling straight to machine code; and programs written in F# or Java are hosted by the .NET framework and the Java Virtual Machine respectively.

For the purpose of this paper, we take a language or dialect being hosted to mean that it utilizes a higher level language than the traditional operating system and C assembly setup for its compilation target and runtime environment. In practice, this often implies JavaScript and its associated browser-provided runtime environments, web applications currently being very much in fashion, but applies equally well to the aforementioned .NET and JVM platforms.

Our contribution In section 2, we present a novel interface for a hosted Haskell dialect to interface with its host language at a high level of abstraction, and describe its implementation for the Haste compiler. [12]

The basic interface is implementable using plain Haskell ’98 with only the Foreign Function Interface extension, and is extensible by the user in the types of data which can be marshalled between Haskell and host language, as well as in how those types are marshalled. It allows for context dependent sanity checking of incoming data from the host language, improving the safety of foreign functions. It generalizes over host languages in the sense that it is usable with any hosted Haskell dialect where the host language supports garbage collection, first class functions, and a construct supporting dynamic code evaluation at runtime such as the eval function of JavaScript, Python, PHP, and others.

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 functions between the host and guest language, and use the GHC generics extension to implement a default marshalling behavior for any Haskell type. We also discuss how to lift the requirement that the host language supports dynamic code evaluation at runtime with a slight modification to the Haskell compiler in use.

For clarity and consistency, we will be using the ECMAScript 6 draft specification [10], from here on referred to simply as JavaScript, as our target platform whenever it is necessary to include code in the host language.

```haskell
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 2: Foreign imports using our FFI

The basic interface is implementable using plain Haskell ’98 with only the Foreign Function Interface extension, and is extensible by the user in the types of data which can be marshalled between Haskell and host language, as well as in how those types are marshalled. It allows for context dependent sanity checking of incoming data from the host language, improving the safety of foreign functions. It generalizes over host languages in the sense that it is usable with any hosted Haskell dialect where the host language supports garbage collection, first class functions, and a construct supporting dynamic code evaluation at runtime such as the eval function of JavaScript, Python, PHP, and others.

Our contribution In section 2, we present a novel interface for a hosted Haskell dialect to interface with its host language at a high level of abstraction, and describe its implementation for the Haste compiler. [12]

The basic interface is implementable using plain Haskell ’98 with only the Foreign Function Interface extension, and is extensible by the user in the types of data which can be marshalled between Haskell and host language, as well as in how those types are marshalled. It allows for context dependent sanity checking of incoming data from the host language, improving the safety of foreign functions. It generalizes over host languages in the sense that it is usable with any hosted Haskell dialect where the host language supports garbage collection, first class functions, and a construct supporting dynamic code evaluation at runtime such as the eval function of JavaScript, Python, PHP, and others.

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 functions between the host and guest language, and use the GHC generics extension to implement a default marshalling behavior for any Haskell type. We also discuss how to lift the requirement that the host language supports dynamic code evaluation at runtime with a slight modification to the Haskell compiler in use.

For clarity and consistency, we will be using the ECMAScript 6 draft specification [10], from here on referred to simply as JavaScript, as our target platform whenever it is necessary to include code in the host language.

```
```
These definitions give rise to a workflow for interacting with host program if and only if:

- a function can be imported from the host language into our Haskell
- the function's return type is convertible into HostAny.

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

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

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

- define the appropriate ToAny and FromAny instances for any custom types, either automatically using the generic default instances as showcased by our motivating example in figure 2, or by defining them manually if a particular host language representation is desired; then
- import arbitrary host language symbols or expressions over those types using the host function.

We let the classic “hello, world” example illustrate the import of simple host language functions using the interface described in figure 3:

```haskell
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:

```haskell
type Time = Double
newtype FrameHandle = FrameHandle HostAny deriving (ToAny, FromAny)
requestFrame :: (Time → IO () → IO FrameHandle)
requestFrame = host "window.requestAnimationFrame"
cancelFrame :: FrameHandle → IO ()
cancelFrame = host "window.cancelAnimationFrame"
```

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

In the rest of section 2, we give an implementation of the basic Haskell '98 interface for the Haste compiler. We then extend it with features requiring some extensions to Haskell '98 — most notably generics and default signatures — in section 4, to arrive at the complete interface presented here.

### 2.2 Implementing marshalling

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

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

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

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

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

```haskell
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

---

**Figure 3:** The programmer’s view of our interface

### 2. An FFI for hosted environments

#### 2.1 The interface

This section describes the programmer’s view of our complete interface and gives examples of its usage. The Haskell formulation of the interface is given in figure 3.

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 language-native representations of arbitrary Haskell values. A data type is then considered to be marshallable if and only if it can be converted to and/or from HostAny.

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:

- its return type is convertible from the host-native HostAny; and
- its return type resides in the IO monad, accounting for the possibility of side effects in host language functions.

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

- define the appropriate ToAny and FromAny instances for any custom types, either automatically using the generic default instances as showcased by our motivating example in figure 2, or by defining them manually if a particular host language representation is desired; then
- import arbitrary host language symbols or expressions over those types using the host function.

We let the classic “hello, world” example illustrate the import of simple host language functions using the interface described in figure 3:

```haskell
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:

```haskell
type Time = Double
newtype FrameHandle = FrameHandle HostAny deriving (ToAny, FromAny)
requestFrame :: (Time → IO () → IO FrameHandle)
requestFrame = host "window.requestAnimationFrame"
cancelFrame :: FrameHandle → IO ()
cancelFrame = host "window.cancelAnimationFrame"
```

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

In the rest of section 2, we give an implementation of the basic Haskell '98 interface for the Haste compiler. We then extend it with features requiring some extensions to Haskell '98 — most notably generics and default signatures — in section 4, to arrive at the complete interface presented here.

### 2.2 Implementing marshalling

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

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

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

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

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

```haskell
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
hosts, we may have different primitive data structures at our disposal.

Fortunately, most host languages of interest support two basic aggregate types, which are sufficient to represent values of any type: arrays and dictionaries.

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

For dictionaries, the conversion is not as clear-cut. Depending on the data we want to convert, the structure of our desired host language representation of two values may well be different even when their guest language 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 user-defined dictionaries.

We will need three basic host language operations: creating a new dictionary, associating a dictionary key with a particular value, and looking up values from dictionary keys. From these we construct two functions to marshal compound Haskell values to and from dictionaries: mkDict and getMember, as shown in figure 4.

This gives us the power to represent any composite or primitive data type with user-defined dictionary keys. Figure 5 shows a possible marshalling for sum and product types using the aforementioned dictionary operations.

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

2.3 Importing functions

Implementing our host function turns out to be slightly trickier than marshalling data between environments. The types of our imported functions need to differ depending on the arity of the imported host language code. This necessitates host returning some variadic function. Fortunately, there is a well known trick to accomplish this, described in [1], 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. As import is a reserved keyword, we have to make do with an alternate spelling.

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

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

```
fromAny x = do
  tag ← getMember x "tag"
  case tag of
    "left" → Left <$> getMember "data"
    "right" → Right <$> getMember "data"
```

Figure 4: Dictionary manipulation

Figure 5: Sums and products using lists and dictionaries

---

```
foreign import ccall
newDict :: IO HostAny

foreign import ccall
set :: HostAny → HostString → HostAny → IO ()

foreign import ccall
get :: HostString → IO HostAny

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

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

---

```
instance (ToAny a, ToAny b) ⇒
  ToAny (Either a b) where
  toAny (Either a b) = fromAny a,
```

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

---

```
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 (a, b) = Import f args =
```

---

```
instance (FromAny a, FromAny b) ⇒
  FromAny (Either a b) where
  fromAny (Either a b) = mkDict [("tag", toAny "left"),
                                 ("data", toAny a)]
```

---

```
instance (FromAny a, FromAny b) ⇒
  FromAny (a, b) where
  fromAny (a, b) = mkDict [("tag", toAny "right"),
                          ("data", toAny b)]
```

---

```
instance (FromAny a, FromAny b) ⇒
  FromAny (a, b) where
  fromAny (a, b) = mkDict [("tag", toAny "right"),
                          ("data", toAny b)]
```
Note the use of a foreign import in our base case. As the application of a foreign function to a foreign list of foreign arguments is clearly, well, a foreign matter, we must call out to the host language for this final step. The list of arguments needs to be converted from [HostAny] to HostAny in order to squeeze it through the vanilla foreign function interface, and the value we get back marshalled back into a proper Haskell value.

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

```
instance (ToAny a, Import b) => Import (a → b) where
  imp0rt f args = 
  Aarg → imp0rt f (toAny arg : args)
```

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

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

host :: Import f ⇒ String → f
    host s = imp0rt f []
    where
      f = eval (toHostString s)
```

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

3. Optimizing for safety and performance

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

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

3.1 Eliminating argument passing overheads

The performance-minded reader may notice something troubling about the implementation of imp0rt: 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 [19], we can eliminate this pointless work in most cases by specializing the host function’s base case instance for different numbers of arguments. In addition to the general apply function we define a series of apply0, apply1, etc. functions, one for each arity we want to optimize function application for. The actual specialization is then a matter of rewriting host calls to use the appropriate application function.

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

Figure 6 gives a new implementation of the base case of the Import class which includes this optimization, replacing the one given in section 2.

3.2 Preventing code injection

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

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

Fortunately, the recent 7.10 release of the GHC compiler gives us the means to eliminate this potential pitfall. The StaticPointers extension, its first incarnation described in [13], 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\(^1\) 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 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 analyze and optimize the resulting code. As discussed in section 5, this effect is very much in evidence when targeting the widely used V8 JavaScript engine.

In the case of JavaScript, it is quite common to run programs through a minifier — a static optimizer with focus on code size — before deployment. Not only do such optimizers suffer the same analytical difficulties as the language runtime itself from the presence of dynamically 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 disadvantage of requiring modifications to the compiler in use, however slight, which increases the interface's overall complexity of implementation.

4. Putting our interface to use

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

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

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 `imp0rt` 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 undeclared instances. Essentially, the loosened restrictions on type class instances allows an `Import` instance to act as a synonym for `FromAny`, allowing host language functions to return functions of any type admissible as an import type by way of the `host` function.

```
instance Import a ⇒ FromAny a where
  fromAny f = return (imp0rt 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 `hsfun.to_host` to convert a Haskell function `f` from `n` `HostAny` arguments to a `HostAny` return value `r` in the IO monad into a host language function which, when applied to `n` host language arguments, calls `f` with those same arguments and returns the `r` returned by `f`.

```
foreign import ccall hsfun.to_host
:: (HostAny → ... → HostAny) → HostFun
```

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

We know that, operationally, `hsfun.to_host` expects a Haskell function as its input, but the types do not agree; we must somehow find a way to pass arbitrary data unchanged to our host language. Fortunately, standard Haskell provides us with a way to do exactly what we want: `StablePointers`.\(^{[20]}\) Note that, depending on the Haskell compiler in use, this use of stable pointers may introduce a space leak. This is discussed further in section 6.2, and an alternative solution is presented.

\(^1\) `safe_host` function is currently not included in our reference implementation, as our work to port the Haste compiler it uses to GHC 7.10 is still in progress.

\(^2\) The main reason for `eval` getting called more than once being unwise inlining directives from the user.
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 [11], to lend us the type level flexibility we need. We begin by defining the Exportable type class first encountered in the type signature of fromAny 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 [11], to lend us the type level flexibility we need. We begin by defining the Exportable type class first encountered in the type signature of fromAny 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.

\[
\text{type family Host a where} \\
\text{Host (a → b) = HostAny → Host b} \\
\text{Host (IO a) = IO HostAny}
\]

\[
\text{class Exportable f where} \\
\text{mkHostFun :: f → Host f}
\]

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 values 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.

\[
\text{instance ToAny a ⇒ Exportable (IO a) where} \\
\text{mkHostFun = fmap toAny}
\]

\[
\text{instance (FromAny a, Exportable b) ⇒ Exportable (a → b) where} \\
\text{mkHostFun f =} \\
\text{mkHostFun . f . fromAny}
\]

\[
\text{instance Exportable f ⇒ ToAny f where} \\
\text{{-# NOINLINE toAny #-}} \\
\text{toAny = unsafePerformIO . fromAny}
\]

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.

fromAny will only be called unsafely when a function is marshalled into a HostAny value and then applied. 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, which is rather unlikely. 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 fromAny. According to [20], the creation of stable pointers residing in the IO monad — the reason for fromAny residing there as well — is to avoid accidentally duplicating the allocation of the stable pointer, something we can avoid by telling the compiler never to inline the function, ever.

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

Marshalling pure functions

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

• The fromAny 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 fromAny will 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 fromAny or a function performing the same conversion on pure functions, depending on the type of function being marshalled.

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

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

Figure 7 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 in [6] do not admit the overlapping type equations required to make pure functions an instance of Exportable.

4.2 Static function exports

Very rarely are users prepared to abandon person-decades of legacy code; to reach these users, the ability to expose Haskell functional-
import Foreign.StablePtr
import System.IO.Unsafe

foreign import ccall
  hsfun_to_host' :: StablePtr a -> HostFun
hsfun_to_host :: Exportable f ⇒ f → IO HostFun
hsfun_to_host f =
  hsfun_to_host' 'fmap' newStablePtr (mkHostFun f)

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

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

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

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

Figure 7: Dynamic function exports implemented on top of our interface

ity to the host language is important. Alas, being implemented as a library, our interface is not capable of foreign export declarations. We can, however, implement a substitute on top of it.

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

Using the function marshalling implemented in section 4.1, implementing export becomes a mere matter of passing a function to the host language, which then arranges for the function to be available in a known, appropriate location.

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

4.3 Generic marshalling

Returning to our motivating example with figure 2, we note a conspicuous absence: the UTCTime instance of FromAny is not defined, yet it is still used by the host function in the definition of getCurrentTime. Although the instance can be defined in a single line of code, it would still be nice if we could avoid the tedium of writing that one line altogether. Thanks to generic programming and default type class instances, we can.

Our implementation of generic marshalling uses GHC generics [17], and associated language extensions — most notably type operators and scoped type variables — making it specific to GHC-based compilers such as Haste and GHCJS [18]. GHC generics allows us to traverse values of any type as though the type was uniformly defined as a tree of sums, products, constants and metadata, such as record selectors or constructor names.

For the sake of brevity, and as the actual syntax of GHC generics is relatively nondescriptive due to its generality, we only give the basic method of our implementation is this paper, and only consider the case of marshalling Haskell values into their host language counterparts. Marshalling in the other direction uses the same basic method, and the complete implementation is available from [15].

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

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

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

• When we reach a constructor argument x of a type t with a ToAny instance, we use that instance to marshal x and return it as a single value: One (toAny x).

• When we reach a record selector metadata node with a selector name n and a child node c, we recursively marshal c and return it paired with its selector name: Tree [(n, toAny c)].

• When we reach a constructor metadata node with a constructor name name and a child node c, we recursively marshal c and call the resulting value c’.

  • c’ is a dictionary, we add an entry to it to mark the value’s constructor name and return the resulting dictionary: Tree ["tag", toAny n : c’].

  • c’ is an empty list, we simply return the constructor name: One (toAny n).

  • c’ is a nonempty list or a single item, we return a new dictionary consisting of the constructor tag and the HostAny encoding of c’:
    Tree ["tag", toAny n], ("data", toAny c’)].

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

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

Note that the case where a tree is merged with a non-tree is undefined. Trees arise only from a use of record selectors. Haskell only allows data constructors where either all arguments have selectors, or none has, meaning that trees and non-trees will never appear in the same product node.
• When we reach a \textit{sum} node, signifying one of several data constructors of a type, we will either have a \textit{left} child or a \textit{right} child. We simply recurse down through whichever child node we have and return the result.

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

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

5. Performance

It must be said, that performance is not a significant motivator for this work. High level features oftentimes come at the price of decreased performance, and calling back and forth between Haskell and the host environment is usually not cheap under the best circumstances. Even so, having a picture of how a particular tool performs vis a vis its alternative in the programmer’s toolbox is a very useful, even essential, thing.

We have measured the performance of an implementation of our interface for the Haste compiler, applying the optimizations described in section 3, against the Haste compiler’s implementation of the vanilla FFI. To establish a baseline for basic marshalling rather than comparing apples to oranges by attempting to include more complex types into our benchmarks, types which may not even be properly marshallable using the vanilla FFI, we have chosen to only benchmark the marshalling of \textit{Doubles}, which map directly to JavaScript’s native \texttt{Number} type.

To get a picture of how our library fares in different scenarios, two microbenchmarks were devised: one which repeatedly applies a foreign function, which performs no computation to make differences in calling overhead stand out as much as possible, in a strict, tight, tail recursive loop, and one which does the same in a higher level loop, constructed from nested calls to \texttt{map} over a list. In order to separately test the overheads of marshalling outbound and inbound data, the benchmarks were created using two variants each: one which ignores the value returned by the foreign function, and one which adds up the results. These benchmarks were then run and timed using the node.js JavaScript interpreter, with the foreign function imported through our library and the vanilla FFI respectively.

While this may not be the most rigorous of performance evaluations, the results are repeatable, and the methodology is enough for our purposes: getting a rough picture of how much speed we are giving up for a more convenient interface. The results for each benchmark are given in table 1 as the ratio of the run time for our library over the run time for the vanilla FFI.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 & Tight loop & High level code \\
\hline
Outbound marshalling only & 0.85 & 1.33 \\
In- and outbound marshalling & 14.52 & 1.76 \\
\hline
\end{tabular}
\caption{Performance numbers, in relation to vanilla FFI}
\end{table}

\textbf{Tight loop, no inbound data} Looking at the performance numbers, our library performs surprisingly well in a highly optimized loop, even edging out the vanilla FFI by a slight margin. The reason for this discrepancy is not entirely clear. The only major difference generated in the JavaScript code generated between the two is the creation and evaluation of one extra thunk, as well as the first iteration of the loop being unrolled, for our interface. This appears to be enough of a change that the JavaScript interpreter is able to optimize that program slightly better than its vanilla FFI equivalent.

\textbf{Loose loop, no inbound data} Putting our library into a higher level context, this small and highly debatable performance advantage disappears. Due to our interface being implemented mainly as a pure Haskell library, the compiler is noticeably worse at figuring out the strictness properties of the program compiled using our library than with the program using the vanilla FFI, leading to some unnecessary creation and evaluation of thunks.

\textbf{Tight loop, with inbound data} Moving on to the benchmarks where we actually marshal incoming data, we see a huge performance hit on inbound data. It turns out, that this is due to the increased safety of our interface. The vanilla foreign function interface happily passes whatever value it gets back into Haskell as a double, potentially wreaking type-unsafe havoc when it turns out that the “double” returned was actually an array of strings containing someone’s shopping list.

In contrast, our library ensures that what is claimed at the type level — that the value coming from the outside world is a \texttt{Double} — is actually true, by passing the marshalled value to the JavaScript built-in function \texttt{Number}. What this benchmark tells us is that checking if something is a number or not is roughly 14 times slower than just copying the same value. By replacing our safe marshalling function with one that mimics the behavior of the vanilla FFI, we see the same performance and generated code as in the \textit{tight loop, no inbound data} case.

\textbf{Loose loop, no inbound data} Moving to a less optimized loop, the percentage of the execution time spend marshalling and calling foreign functions drops drastically, as we can see by the performance hit of our library being much smaller than for the tight, optimized loop. Analyzing the generated code, this is again due to the extra safety net, with an additional slight performance disadvantage from the return value being wrapped in a thunk, despite obviously (to a human) being strict coming from the host language; the malus to strictness analysis incurred by our library in action again.

\textbf{Performance verdict: acceptable} Judging by these numbers, disregarding the additional cost incurred by the additional safety measures, the performance of our library is quite acceptable, ranging from slightly faster than the vanilla FFI to at most twice as slow. Considering that calls between Haskell and the host environment are usually relatively few and far between, this performance difference should be negligible. For code which has no choice but to make a large number of calls to low level host language functions in performance critical loops, using the vanilla FFI alongside our library is usually enough, and calling back and forth between Haskell and the host environment is usually not cheap under the best circumstances. Even so, having a picture of how a particular tool performs vis a vis its alternative in the programmer’s toolbox is a very useful, even essential, thing.

While the additional performance cost incurred from the additional safety may be unwanted when the user is completely sure that a particular function will never, ever return anything but an actual
Number, said user is free to write an UnsafeDouble wrapper with an appropriately fast and loose fromAny instance to sidestep the (hopefully) unnecessary safety measures.

6. Discussion

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

6.1 fromAny type level expressiveness

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

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

It is then simpler, not to mention more performant, to trust that the foreign code in question is well behaved — something we must do anyway since we automatically rely on its correctness by allowing our program to depend upon it — and throw that exception immediately on conversion failure rather than taking a detour via error values, should this trust prove to be misplaced.

6.2 Limitation to garbage collected host languages

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

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

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

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

6.3 Restricting imports to the IO monad

The interface presented in this paper does not support importing pure functions; any function originating in the host language must be safely locke 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.

6.4 Continuation-based blocking in non-concurrent environments

A particularly neat feature of the foreign function interface employed by the GHCJS compiler is the ability for foreign host code to suspend execution while waiting for an event to occur, even
though its JavaScript host environment is devoid of any concurrency support. [18] This is accomplished by giving imported functions an extra parameter: a continuation to be called upon completion of the foreign operation to resume execution of the Haskell program, instead of simply returning like a “normal” JavaScript function would.

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

Instead, this functionality can be implemented on top of our interface without much difficulty using a construct dubbed the “poor man’s concurrency monad” [7]; a monad implementing cooperative multitasking with blocking synchronization variables in non-concurrent environments.

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 specialization: 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 over as wide a range of Haskell dialects as possible.

**Idris: host-parametric FFI** Idris is a dependently typed, Haskell-like language with backends for several host environments, JavaScript being one of them. [3] 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 parameterized over the target environment. This ensures that Idris code written specifically for a native environment is not accidentally called from code targeting JavaScript and vice versa.

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

**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”. [9] While the two are very similar in syntax, allowing users to import typed strings of host language code, Fay’s solution is highly specialized. The compiler takes a heavy hand in the marshalling and import functionality, parsing the host language code and performing certain substitutions on it. While marshalling of arbitrary types is available, this marshalling is not easily controllable by the user, but follows a sensible but fixed format determined by the compiler. This approach makes sense, as the interface is designed to support the Fay language and compiler alone, but differs from our work which aims to create a more generally applicable interface.

**GHCJS: JavaScriptFFI** The GHCJS Haskell-to-JavaScript compiler [18] utilizes 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 when 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. [8] 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 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.

**Quasi-quotes** Quasi-quotes represent another, more radically different, approach to the problem of bridging with a host language. [16] Allowing for the inline inclusion of large snippets of foreign code, 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. [5]

Unfortunately, this usefulness comes at the price of a rather complex implementation. Not only is complex templating support from the compiler required, but implementations need to be able to parse the host language as well. A price in implementation effort that may not always be worth the extra expressive power it buys.

7. Conclusions and future work

**Future work** By combining two optimizations 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 lift the restriction that our host language support dynamic code evaluation: if all foreign imports are statically known, and we are able to eliminate eval calls for all statically known functions, it follows that we are able to eliminate all eval calls. While the actual implementation of this idea has yet to be worked out, guaranteeing the complete absence of eval from the generated host code would remove the restriction that our host language supports dynamic code evaluation at runtime,
notably making our interface implementable on recent versions of
the Java Virtual Machine. Implementing this interface for the Java
Virtual Machine, with the prerequisite Haskell-to-JVM compiler,
would lend further weight to our assertion that this interface is both
both useful and broadly implementable.
Out interface does not currently catch and marshal host language
exceptions, but requires foreign language code to take care of any
exceptions. While the actual implementation is quite specific to a
particular host environment, automatically converting exceptions
would be a useful feature even so. Investigating the degree to
which this feature could be implemented in a host platform agnostic
manner would be a possible extension of this work.
Due to the hard requirement that our host language be garbage
collected, our interface is not currently applicable in a C context.
This is unfortunate, as C-based host environments are still by far
the most common for Haskell programs. It may thus be worthwhile
to investigate the compromises needed to lift the garbage collection
requirement from potential host environments.

Conclusions

We have presented the design of a novel, portable
and platform agnostic foreign function interface for hosted Haskell
dialects. We have also given a number of optimizations, improving
the performance and safety of our interface and lightening the re-
strictions placed on the host environment, and implemented our in-
terface as a library for the Haste Haskell-to-JavaScript compiler. Fi-
nally, we have used this library to further extend our marshalling ca-
pabilities to cover functions, as well as generic default marshalling
for arbitrary data types, contrasted our approach with a variety of
existing foreign function interfaces, and demonstrated that our li-
brary is relatively performant.
While our interface is currently not applicable to Haskell imple-
mentations targeting low level, C-like environments, it brings sig-
ificant reductions in boilerplate code and complexity for users
needing to interface their Haskell programs with their correspond-
ing host environment in the space where it is applicable: Haskell
implementations for high level target platforms.
Our reference implementation is shipped in both source code and
binary form as part of the Haste distribution, available from the
Haste Language website at [15].

Acknowledgments

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

References

Available from http://hackage.haskell.org/package/base-4.8.0.0/
docs/Text-Printf.html, 2015-04-01.
Interface 1.0: An Addendum to the Haskell 98 Report.
on Haskell (pp. 119-120). ACM.
Associated type synonyms. In ACM SIGPLAN Notices (Vol. 40, No. 9,
pp. 241-253). ACM.
Functional Programming, 9(03), 313-323.
Building JavaScript applications with Haskell. In Implementation and
Application of Functional Languages (pp. 37-52). Springer Berlin
Heidelberg.
S. (2014). Closed type families with overlapping equations. ACM
SIGPLAN Notices, 49(1), 671-683.
centric programming model for type safe web applications. In Proceed-
ings of the 2014 ACM SIGPLAN symposium on Haskell (pp. 79-89).
ACM.
Towards Haskell in the cloud. In ACM SIGPLAN Notices (Vol. 46, No.
12, pp. 118-129). ACM.
http://www.haskell.org/ghc/docs/7.10.1/users_guide.pdf, 2015-
04-01.
2015-05-14.
quasiquoting for haskell. In Proceedings of the ACM SIGPLAN
workshop on Haskell workshop (pp. 73-82). ACM.
deriving mechanism for Haskell. In Haskell 2010, Baltimore, MD, USA,
[18] Nazarov, V., Mackenzie, H., & Stegeman, L. "GHCJS Haskell to
JavaScript Compiler". Available from https://github.com/ghcjs/
by the rules: rewriting as a practical optimisation technique in GHC.
In Haskell workshop (Vol. 1, pp. 203-233).
Improving Haskell’s foreign language interface. In Glasgow Functional
Programming Workshop Draft Proceedings, Ayr, Scotland.
haskell.org/package/time, 2015-04-01.