Foreign Exchange at Low, Low Rates

A lightweight FFI for web-targeting Haskell dialects

Anton Ekblad Chalmers University of Technology
antonek@chalmers.se

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 optimized 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.

CCS Concepts

•**Information systems** → **Web applications;** •**Software and its engineering** → **Functional languages;** *Runtime environments;*

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 [19]. 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 [9] and GHCJS [13] – 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 [3] to touch them. While the example given in figure 1 certainly works when compiled with either Haste or GHCJS, it is not something the user would like to write.

```
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
```

Figure 1: Foreign imports using the vanilla Foreign Function Interface

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

IFL '15 September 14-16, 2015, Koblenz, Germany

ACM ISBN 978-1-4503-4273-5/15/09.

DOI: http://dx.doi.org/10.1145/2897336.2897338

```
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

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

Contrasting this with the standard FFI code from figure 1:

- 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 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 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 1, 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 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-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

```
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 3: The programmer's view of our interface

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.

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 3:

```
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.

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

[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 4: Marshalling arrays and dictionaries

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 4.
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 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 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

```
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 5: Sums and products using lists and dictionaries

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" [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.

```
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:

```
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`.

```
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 2

3. OPTIMIZING FOR SAFETY AND PERFORMANCE

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

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

3.1 Eliminating argument passing overheads

The performance-minded reader may notice something troubling about the implementation of `import_`: the construction of an intermediate list of 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* [15], we can eliminate this pointless work in most cases by specializing the `host` function's base case instance for different numbers of arguments. In addition to the general `apply` function we define a series of `apply0`, `apply1`, etc. functions, one for each arity we want to optimize function application for. The actual specialization is then a matter of rewriting `host` calls to use the appropriate application function.

Figure 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

```
{-# 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 6: Specializing the `host` base case

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

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

Fortunately, the recent 7.10 release of the GHC compiler, 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 [10], introduces the `static` keyword, which is used to create values of type `StaticPtr` from closed expressions. Attempting to turn any expression which is not known at compile time into a `StaticPtr` yields a compiler error.

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

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

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

3.3 Eliminating `eval`

Relying on `eval` to produce our functions allows us to implement our interface in pure Haskell '98 without modifying the Haskell compiler in question, making the interface easy to understand, implement and maintain. However, there are reasons why it may be

in the implementor's best interest to forgo a small bit of that simplicity.

The actual call to `eval` does not meaningfully impact performance: it is generally only called once per import, the resulting function object cached thanks to lazy evaluation.[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 optimize the resulting code. As discussed in section 5, this effect is very much in evidence when targeting the widely used V8 JavaScript engine.

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

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

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

Passing functions to foreign code.

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

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

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

```
foreign import 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: `StablePointer`s [17]. 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.

```
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 [8], 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 behavior of our function marshalling.

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

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

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

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

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

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

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

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

When a function is marshalled into a `HostAny` value and subsequently applied, `fromAny` will be applied unsafely to each of the marshalled function's arguments. There are two cases when this can happen: either the marshalled function is called from the host language, or it is marshalled back into Haskell and then applied. In the former case, the time of the call is trivially well-defined assuming that our target language is not lazy by default. In the latter case, the time of the call is still well-defined, as our interface only admits importing functions in the `IO` monad.

Slightly more troubling is the use of `unsafePerformIO` in conjunction with `hsfun_to_host`. According to [17], 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 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 [5] 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 7: 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 forein 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 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. 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 [11] or Template Haskell [18], 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 [14]. This default instance provides the final piece of the puzzle required to use the interface as presented in figure 2.

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 appropraite action.

`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 8.

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 over-

```
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 8: Marshalling JavaScript exceptions

head 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 2 against the equivalent implementation given in figure 1, 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 optimize and analyse for strictness.
The resulting programs, compiled with version 0.5.4.2 of the Haste compiler which incorporates all the optimizations described in section 3, 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 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 optimized and less optimized 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.

	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 1: Execution times as fractions of FFI execution times

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 favorably 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 optimization described in section 3.3 does not impact performance measurably in these benchmarks. If any performance benefit is to be had from this optimization, 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 unfavorable 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 behavior 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 fulfill 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.

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

data FakeStablePtr a
fakeStablePtr :: a → FakeStablePtr a

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

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

```
data FakeStablePtr a = Fake !a

fakeStablePtr = Fake
```

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

6.4 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 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, maximizing 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 [2]. 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" [7]. 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 [13] 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 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 [6]. 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 [16]. How-

ever, 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 [12]. 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 [4].
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 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 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 optimizations, 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. REFERENCES

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

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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