Project H: Programming R in Haskell

PRELIMINARY DRAFT

Mathieu Boespflug Facundo Domínguez
Alexander Vershilov

Tweag I/O

Allen Brown

Amgen

Abstract

A standard method for augmenting the "native" set of libraries available within any given programming environment is to extend this set via a foreign function interface provided by the programming language. In this way, by exporting the functionality of external libraries via *binding modules*, one is able to reuse libraries without having to reimplement them in the language *du jour*.

However, *a priori* bindings of entire system libraries is a tedious process that quickly creates an unbearable maintenance burden. We demonstrate an alternative to monolithic and imposing binding modules, even to make use of libraries implemented in a special-purpose, dynamically typed, interpreted language. As a case study, we present H, an R-to-Haskell interoperability solution making it possible to program all of R, including all library packages on CRAN, from Haskell, a general-purpose, statically typed, compiled language. We demonstrate how to do so efficiently, without marshalling costs when crossing language boundaries and with static guarantees of well-formation of expressions and safe acquisition of foreign language resources.

Keywords R, Haskell, foreign function interface, quasiquotation, language embedding, memory regions

1. Introduction

The success or failure in the industry of a programming language within a particular problem domain is often predicated upon the availability of a sufficiently plethoric set of good quality libraries relevant to the domain. Libraries enable code reuse, which ultimately leads to shorter development cycles. Yet business and regulatory constraints may impose orthogonal requirements that not all programming languages are able to satisfy.

Case in point: at Amgen, we operate within a stringent regulatory environment that requires us to establish high confidence as to the correctness of the software that we create. In life sciences, it is crucial that we aggressively minimize the risk that any bug in our code, which could lead to numerical, logical or modelling errors with tragic consequences, goes undetected.

[Copyright notice will appear here once 'preprint' option is removed.]

We present a method to make available any foreign library without the overheads typically associated with more traditional approaches. Our goal is to allow for the seamless integration of R with Haskell — invoking R functions on Haskell data and *vice versa*.

Foreign Function Interfaces The complexity of modern software environments makes it all but essential to interoperate software components implemented in different programming languages. Most high-level programming languages today include a *foreign function interface (FFI)*, which allows interfacing with lower-level programming languages to get access to existing system and/or purpose-specific libraries [2, 9]. An FFI allows the programmer to give enough information to the compiler of the host language to figure out how to *invoke* a foreign function included as part of a foreign library, and how to *marshal* arguments to the function in a form that the foreign function expects. This information is typically given as a set of bindings, one for each function, as in the example below:

```
{-# LANGUAGE ForeignFunctionInterface #-}
module Example1 (getTime) where
import Foreign
import Foreign.C

#include <time.h>

data TimeSpec = TimeSpec
  { seconds     :: Int64
  , nanoseconds :: Int32
  }

foreign import ccall "clock_gettime"
  c_clock_gettime :: ClockId → Ptr TimeSpec → IO CInt

getTime :: ClockId → IO TimeSpec
getTime cid = alloca $ λts. do
  throwErrnoIfMinus1_ "getTime" $
    c_clock_gettime cid ts
  peek ts
```

In the above, c_clock_gettime is a binding to the `clock_gettime()` C function. The API conventions of C functions are often quite different from that of the host language, so that it is convenient to export the wrapper function getTime rather than the binding directly. The wrapper function takes care of converting from C representations of arguments to values of user defined data types (performed by the peek function, not shown), as well as mapping any foreign language error condition to a host language exception.

Binding generators These bindings are tedious and error prone to write, verbose, hard to read and a pain to maintain as the API of the underlying library shifts over time. To ease the pain, over the years,

binding generators have appeared [1], in the form of pre-processors that can parse C header files and automate the construction of binding wrapper functions and argument marshalling. However, these tools:

1. do not alleviate the need for the programmer to repeat in the host language the type of the foreign function;
2. add yet more complexity to the compilation pipeline;
3. being textual pre-processors, generate code that is hard to debug;
4. are necessarily limited in terms of the fragments of the source language they understand and the types they can handle, or repeat the complexity of the compiler to parse the source code.

Point (1) above is particularly problematic, because function signatures in many foreign libraries have a knack for evolving over time, meaning that bindings invariably lag behind the upstream foreign libraries in terms of both the versions they support, and the number of functions they bind to.

Moreover, such binding generators are language specific, since they rely on intimate knowledge of the foreign language in which the foreign functions are available. In our case, the foreign language is R, which none of the existing binding generators support. We would have to implement our own binding generator to alleviate some of the burden of working with an FFI. But even with such a tool in hand, the tedium of writing bindings for all standard library functions of R, let alone all functions in all CRAN packages, is but a mildly exciting prospect. One would need to define a monolithic set of bindings (i.e. a *binding module*), for *each* R package. Because we cannot anticipate exactly which functions a user will need, we would have little recourse but to make these bindings as exhaustive as possible.

Rather than *bind* all of R, the alternative is to *embed* all of R. Noting that GHC flavoured Haskell is a capable meta-programming environment, the idea is to define code generators which, at each call site, generates code to invoke the right R function and pass arguments to it using the calling convention that it expects. In this way, there is no need for *a priori* bindings to all functions. Instead, it is the code generator that produces code spelling out to the compiler exactly how to perform the R function call – no binding necessary.

It just so happens that the source language for these code generators is R itself. In this way, users of H may express invocation of an R function using the full set of syntactical conveniences that R provides (named arguments, variadic functions, *etc.*), or indeed write arbitrary R expressions. R has its own equivalent to `clock_gettime()`, called `Sys.time()`. With an embedding of R in this fashion, calling it is as simple as:

```
printCurrentTime = do
  now ← ⟦r| Sys.time() ⟧
  putStrLn ("The current time is: " ++ fromSEXP now)
```

The key syntactical device here is *quasiquotes* [7], which allow mixing code fragments with different syntax in the same source file — anything within an ⟦r| ... ⟧ pair of brackets is to be understood as R syntax.

Contributions In this paper, we advocate for a novel approach to programming with foreign libraries, and illustrate this approach with the first complete, high-performance tool to access all of R from a statically typed, compiled language. We highlight the difficulties of mixing and matching two garbage collected languages that know nothing about each other, and how to surmount them by bringing together existing techniques in the literature for safe memory management [6]. Finally, we show how to allow optionally ascribing precise types to R functions, as a form of compiler-checked documentation and to offer better safety guarantees.

Outline The paper is organized as follows. We will first walk through typical uses of H, before presenting its overall architecture (Section 2). We delve into a number of special topics in later sections, covering how to represent foreign values efficiently in a way that still allows for pattern matching (Section 3.1), optional static typing of dynamically typed foreign values (Section 3.2), creating R values from Haskell (Section 3.3) and efficient memory management in the presence of two separately managed heaps with objects pointing to arbitrary other objects in either heaps (Section 3.4). We conclude with a discussion of the overheads of cross language communication (Section 4) and an overview of related work (Section 5).

2. H walkthrough and overall architecture

2.1 Foreign values

Foreign functions act on *values*, for which presumably these foreign functions know the representation in order to compute with them. In the specific case of R, *all* values share a common structure. Internally, R represents every entity that it manipulates, be they scalars, vectors, uninterpreted term expressions, functions or external resources such as sockets, as pointers to a `SEXPREC` structure, defined in C as follows:

```
typedef struct SEXPREC {
    SEXPREC_HEADER;
    union {
        struct primsxp_struct primsxp;
        struct symsxp_struct symsxp;
        struct listsxp_struct listsxp;
        struct envsxp_struct envsxp;
        struct closxp_struct closxp;
        struct promsxp_struct promsxp;
    } u;
} SEXPREC, *SEXP;
```

Each variant of the union struct corresponds to a different form of value. However, no matter the form, all values at least share the same header (called `SEXPREC_HEADER`). The type of pointers to `SEXPREC`s is abbreviated as `SEXP`. In order to invoke functions defined in R then, we simply need a way to construct the right `SEXPREC` representing that invocation, and then have R interpret that invocation. We will cover how to do so in Section 2.2, but for now we do need to define in Haskell what a `SEXP` is:

```
data SEXPREC
type SEXP = Ptr SEXPREC
```

2.2 Interoperating scripting languages

R source code is organized as a set of *scripts*, which are loaded one by one into the R interpreter. Each statement in a each script is evaluated in-order and affects the global environment maintained by the R interpreter, which maps symbols to values. In its simplest form, H is an *interactive environment* much like R, with a global environment altered by the in-order evaluation of statements.

The central and most general mechanism by which H allows interoperating with R is quasiquotation. A *quasiquote* is a partial R script — that is, a script with "holes" in it that stand in for as of yet undetermined portions. An example quasiquote in Haskell of an R snippet is:

```
⟦r| function(x) x + 1 ⟧
```

This quasiquote is *ground*, in that it does not contain any holes (called *antiquotes*), but one can also antiquote inside a quasiquote:

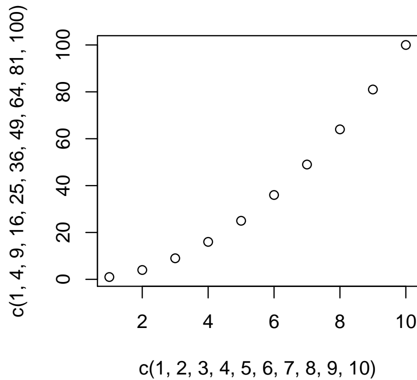

Figure 1. Output of ⟦r| `plot(xs_hs, ys_hs)` ⟧. The data is generated from Haskell. R draws the plot.

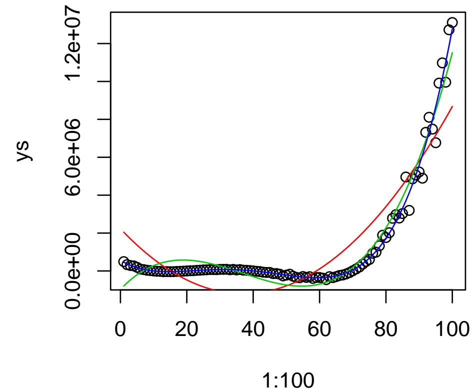

Figure 2. Fitting polynomial models of increasing degree ($n = \{2, 3, 4\}$) to a set of points in Haskell. R fits the models.

```
let y = mkSEXP 1
in ⟦r| function(x) x + y_hs ⟧
```

By convention, any symbol with a `_hs` suffix is treated specially (see Section 2.5). It is interpreted as a reference to a Haskell variable defined somewhere in the ambient source code. That is, any occurrence of a symbol of the form `x_hs` does not denote a variable of the object language — it is an antiquote referring to variable x in the host language. Given any quasiquote, it is possible to obtain a full R script, with no holes in it, by *splicing* the value of the Haskell variables into the quasiquote, in place of the antiquotes.

At a high-level, H is a desugarer for quasiquotes implemented on top of a Haskell interactive environment, such as GHCi [4]. It defines how to translate a quasiquotation into a Haskell expression. Just as R includes an interactive environment, H includes an interactive environment, where the input is a sequence of Haskell expressions including quasiquoted R code snippets, such as in the following session, where we plot part of the quadratic function, directly from the Haskell interactive prompt:

```
H⟩ let xs = [1 .. 10] :: [Double]
H⟩ let ys = [x^2 | x ← xs]
H⟩ result ← ⟦r| as.character(ys_hs) ⟧
H⟩ H.print result
[1] "1" "4" "9" "16" "25" "36" "49" "64" ...
H⟩ ⟦r| plot(xs_hs, ys_hs) ⟧
⟨graphic output (See Figure 1)⟩
```

Now say that we are given a set of random points, roughly fitted by some non-linear model. For the sake of example, we can use points generated at random along a non-linear curve by the following Haskell function:

```
import System.Random.MWC
import System.Random.MWC.Distributions

generate :: Int32 → IO Double
generate ix =
  withSystemRandom ∘ asGenIO $ λgen.
    let r = (x − 10) ∗ (x − 20) ∗ (x − 40) ∗ (x − 70)
            + 28 ∗ x ∗ (log x)
    in do v ← standard gen
    return $ r ∗ (1 + 0.15 ∗ v)
    where x = fromIntegral ix
```

As before, take a set of coordinates:

```
H⟩ ⟦r| xs <- c(1:100) ⟧
H⟩ ⟦r| ys <- mapply(generate_hsfun, xs) ⟧
```

`generate_hsfun` is a *function splice* — just like any other splice, except that the spliced value is higher-order, i.e. a function. R's `mapply()` applies the Haskell function to each element of `xs`, yielding the list `ys`.

Our goal is to ask R to compute estimates of the parameters of polynomial models of increasing degree, with models of higher degree having a higher chance of fitting our dataset well. The R standard library provides the `nls()` function to compute the non-linear least-squares estimate of the parameters of the model. For example, we can try to fit a model expressing the relation between `ys` to `xs` as a polynomial of degree 3:

```
H⟩ ⟦r| P3 <- ys ~ a3*xs**3 + a2*xs**2 + a1*xs + a0 ⟧
H⟩ ⟦r| initialPoints <- list(a0=1,a1=1,a2=1,a3=1) ⟧
H⟩ ⟦r| model3 <- nls(P3, start=initialPoints) ⟧
```

As the degree of the model increases, the residual sum-of-squares decreases, to the point where in the end we can find a polynomial that fits the dataset rather well, as depicted in Figure 2, produced with the following code:

```
⟦r| plot(xs,ys) ⟧
⟨graphic output (See Figure 2)⟩
⟦r| lines(xs,predict(model2), col = 2) ⟧
⟦r| lines(xs,predict(model3), col = 3) ⟧
⟦r| lines(xs,predict(model4), col = 4) ⟧
```

2.3 Scripting from compiled modules

While an interactive prompt is extremely useful for exploratory programming, writing a program as a sequence of inputs for a prompt is a very imperative style of programming with limited abstraction facilities. Fortuntaly, H is also a library. Importing the library brings in scope the necessary definitions in order to embed quasiquotes such as the above in modules of a compiled program.

Behind the scenes, the H library hosts an *embedded instance* of the R interpreter, available at runtime. As in the interactive environment, this embedded instance is stateful. It is possible to mutate the global environment maintained by the interpreter, say by introducing a new top-level definition. Therefore, interaction with this embedded instance must be sequential. In order to enforce sequential access to the interpreter, we introduce the R monad and make all code that ultimately calls into the interpreter *actions* of

the R monad. As a first approximation, the R monad is a simple wrapper around the IO monad (but see Section 3.4)[1].

$$\textbf{newtype}\ \mathsf{R}\ a = \mathsf{R}\ (\mathsf{IO}\ a)$$
$$\mathsf{withEmbeddedR} :: \mathsf{R}\ a \to \mathsf{IO}\ a$$

withEmbeddedR first spawns an embedded instance, runs the provided action, then finalizes the embedded instance. There is no other way to run the R monad.

2.4 Rationale

R is a very dynamic language, allowing many code modifications during runtime, such as rebinding of top-level definitions, super assignment (modifying bindings in parent scopes), (quasi-)quotation and evaluation of expressions constructed dynamically. The R programming language is also so-called ”latently typed” - types are checked during execution of the code, not ahead of time. Many of these features are not compiler friendly.

Haskell, by contrast, is designed to be much easier to compile. This means that not all R constructs and primitives can be readily mapped to statically generated Haskell code with decent performance. In particular, top-level definitions in Haskell are never dynamically rebound at runtime: a known function call is hence often a direct jump, rather than incurring a dynamic lookup in a symbol table (the environment).

Much of the dynamic flavour of R likely stems from the fact that it is a scripting language. The content of a script is meant to be evaluated in sequential order in an interactive R prompt. The effect of loading multiple R scripts at the prompt is in general different depending on the order of the scripts.

Central to the design of Haskell, by contrast, is the notion of separately compilable units of code, called *modules*. Modules can be compiled separately and in any order (provided some amount of metadata about dependencies). Contrary to R scripts, the order in which modules are loaded into memory is non-deterministic.

For this reason, in keeping to a simple solution to interoperating with R, we choose to devolve as much processing of R code as possible to an embedded instance of the R interpreter and retain the notion of global environment that R provides. This global environment can readily be manipulated from an interactive environment such as GHCi [4]. In compiled modules, access to the environment as well as encapsulation of any effects can be mediated through a custom monad, the R monad presented in Section 2.3.

2.5 Runtime reconstruction of expressions

Quasiquotes are purely syntactic sugar that are expanded at compile time. They have no existence at runtime. A quasiquote stands for an R expression, which at runtime we therefore have to reconstruct. In other words, the expansion of a quasiquote is code that generates an R expression. For a ground quasiquote whose content is R expression S, we construct a Haskell expression E, such that

$$\texttt{R_Parse}(S) = E.$$

This law falls out as a special case of a more general law about antiquotation: for any substitution σ instantiating each antiquoted variable in S with some SEXP, we should have that

$$\texttt{R_Parse}(S)\sigma = E\sigma.$$

That is, the abstract syntax tree (AST) constructed at runtime (right hand side) should be equivalent to that returned by `R_parse()` at compile time (left hand side). The easiest way to ensure this property is to simply use the R parser itself to identify what AST we need to build. Fortunately, R does export its parser as a standalone function, making this possible. Note that we only call the parser at compile time — reconstructing the AST at runtime programmatically is much faster than parsing.

The upside of reusing R's parser when expanding quasiquotes is that we get support for all of R's syntax, for free, and avoid a potential source of bugs. The flipside is that we cannot reliably extend R's syntax with meta-syntactic constructs for antiquotation. We must fit within R's existing syntax. It is for this reason that antiquotation does not have a dedicated syntax, but instead usurps the syntax of regular R variables.

[1] This definition does not guarantee that R actions will only be executed when the associated R interpreter instance is extant. See Section 3.4 for a fix.

3. Special topics

3.1 A native view of foreign values

Programming across two languages typically involves a tradeoff: one can try shipping off an entire dataset and invoking a foreign function that does all the processing in one go, or keep as much of the logic in the host language and only call into foreign functions punctually. For example, mapping an R function `frobnicate()` over a list of elements might be done entirely in R, on the whole list at once,

$$\texttt{H}\rangle\ \ ys \leftarrow [\![\mathsf{r}|\ \texttt{mapply(frobnicate, xs_hs)}\ ]\!]$$

or elementwise, driven from Haskell,

$$\texttt{H}\rangle\ \ ys \leftarrow \mathsf{mapM}\ (\lambda x.\,[\![\mathsf{r}|\ \texttt{frobnicate(x_hs)}\ ]\!])\ xs$$

The latter style is often desirable — the more code can be kept in Haskell the safer, because more code can be type checked statically.

The bane of language interoperability is the perceived cost of crossing the border between one language to another during execution. Any significant overhead incurred in passing arguments to a foreign function and transferring control to it discourages tightly integrated programs where foreign functions are called frequently, such as in the last line above. Much of this cost is due to marshalling values from the native representation of data, to the foreign representation, and back.

By default, and in order to avoid having to pay marshalling and unmarshalling costs for each argument every time one invokes an internal R function, we represent R values in exactly the same way R does, as a pointer to a SEXPREC structure (defined in `R/Rinternals.h`). This choice has a downside, however: Haskell's pattern matching facilities are not immediately available, since only algebraic datatypes can be pattern matched.

HExp is R's SEXP (or `*SEXPREC`) structure represented as a (generalized) algebraic datatype. Each `SEXPREC` comes with a "type" tag the uniquely identifies the layout (one of `primsxp_struct`, `symsxp_struct`, *etc.* as seen in Section 2.1). See Figure 3 for an excerpt of the R documentation enumerating all possible type tags[2]. A simplified definition of HExp would go along the lines of Figure 4. Notice that for each tag in Figure 3, there is a corresponding constructor in Figure 4.

For the sake of efficiency, we do not use HExp as the basic datatype that all H generated code expects. That is, we do not use HExp as the universe of R expressions, merely as a *view*. We introduce the following *view function* to locally convert to a HExp, given a SEXP from R.

$$\mathsf{hexp} :: \mathsf{SEXP} \to \mathsf{HExp}$$

The fact that this conversion is local is crucial for good performance of the translated code. It means that conversion happens at each use site, and happens against values with a statically known form. Thus we expect that the view function can usually be inlined, and the

[2] In R 3.1.0, there are 23 possible tags.

NILSXP There is only one object of type NILSXP, R_NilValue, with no data.

SYMSXP Pointers to the PRINTNAME (a CHARSXP), SYMVALUE and INTERNAL. (If the symbol's value is a .Internal function, the last is a pointer to the appropriate SEXPREC.) Many symbols have the symbol value set to R_UnboundValue.

LISTSXP Pointers to the CAR, CDR (usually a LISTSXP or NILSXP) and TAG (a SYMSXP or NILSXP).

CHARSXP LENGTH, TRUELENGTH followed by a block of bytes (allowing for the nul terminator).

REALSXP LENGTH, TRUELENGTH followed by a block of C doubles.

...

Figure 3. Extract from the R documentation enumerating all the different forms that values can take.

short-lived HExp values that it creates is compiled away by code simplification rules applied by GHC. Notice how HExp as defined in Figure 4 is a *shallow view* — the fields of each constructor are untranslated SEXP's, not HExp's. In other words, a HExp value corresponds to the one-level unfolding of a SEXP as an algebraic datatype. The fact that HExp is not a recursive datatype is crucial for performance. It means that the hexp view function can be defined non-recursively, and hence is a candidate for inlining[3].

In this manner, we get the convenience of pattern matching that comes with a *bona fide* algebraic datatype, but without paying the penalty of allocating long-lived data structures that need to be converted to and from R internals every time we invoke internal R functions or C extension functions.

Using an algebraic datatype for viewing R internal functions further has the advantage that invariants about these structures can readily be checked and enforced, including invariants that R itself does not check for (e.g. that types that are special forms of the list type really do have the right number of elements). The algebraic type statically guarantees that no ill-formed type will ever be constructed on the Haskell side and passed to R.

We also define an inverse of the view function:

```
unhexp :: HExp → SEXP
```

3.2 "Types" for R

3.2.1 Of types, classes or forms

Haskell is a statically typed language, whereas R is a dynamically typed language. However, this apparent mismatch does not cause any particular problem in practice. This is because the distinction between "statically typed" languages and "dynamically typed" languages is largely artificial, stemming from the conflation of two distinct concepts: that of a *class* and that of a *type* [5].

The prototypical example of a type with multiple classes of values is that of complex numbers. There is *one* type of complex numbers, but *two* (interchangeable) classes of complex numbers: those in rectangular coordinates and those in polar coordinates. Both classes represent values of the same type. Harper further points out:

> Crucially, the distinction between the two classes of complex number is dynamic in that a given computation may result in a number of either class, according to convenience or convention. A program may test whether a complex number is in polar or rectangular form, and we can form data structures such as sets of complex numbers in which individual elements can be of either form.

[3] The GHC optimizer never inlines recursive functions.

```
data HExp
  = Nil                            -- NILSXP
  | Symbol SEXP SEXP SEXP          -- SYMSXP
  | List SEXP SEXP SEXP            -- LISTSXP
  | Char Int32 (Vector Word8)      -- CHARSXP
  | Real Int32 (Vector Double)     -- REALSXP
  | ...
```

Figure 4. Untyped HExp view.

Hence what R calls "types" are better thought of as "classes" in the above sense. They correspond to *variants* (or *constructors*) of a single type in the Haskell sense. R is really a unityped language.

We call the type of all the classes that exist in R the *universe* (See Section 3.1). Each variant of the union field in the SEXPREC structure defined in Section 2.1 corresponds to a class in the above sense. The SEXPREC structure *is* the universe.

Because "class" is already an overloaded term in both R and in Haskell, in the following we use the term *form* to refer to what the above calls a "class".

Some R functions expect a large number of arguments. It is not always clear what the usage of those functions is. It is all too easy to pass a value of the wrong form as an argument, or provide too many arguments, or too few. R itself cannot detect such conditions until runtime, nor is it practical to create a static analysis for R to detect them earlier, given the permissive semantics of the language. However, some information about the expected forms for arguments is given in R's documentation for practically every function in the standard library. It is often useful to encode that information from the documentation in machine checkable form, in such a way that the Haskell compiler can bring to bear its own existing static analyses to check for mismatches between formal parameters and actual arguments.

3.2.2 Form indexed values

To this end, in H we have *form indexed* SEXPs. The actual definition of a SEXP in H is:

```
newtype SEXP s (a :: SEXPTYPE)
  = SEXP (PTR SEXPREC)
```

The a parameter refers to the form of a SEXP (See Section 3.4 for the meaning of the s type parameter). In this way, a SEXP of form REALSXP (meaning a vector of reals), can be ascribed the type SEXP s R.Real, distinct from SEXP s R.Closure, the type of closures. These types are all of kind SEXPTYPE, a datatype promoted to a kind[4]. Each inhabitant of the SEXPTYPE kind corresponds to a type tag as seen in Figure 3.

When some given function is used frequently throughout the code, it is sometimes useful to introduce a wrapper for it in Haskell, ascribing to it a particular type. For example, the function that parses source files can be written as[5]:

```
import qualified Foreign.R.Type as R

parse :: SEXP s 'R.String   -- Filename of source
      → SEXP s 'R.Int       -- Number of expressions to parse
      → SEXP s 'R.String    -- Source text
```

[4] Using GHC's -XDataKind language extension.

[5] The ' is an artifact of the -XDataKinds extension, useful for disambiguation. It is not strictly necessary, but it is good practice to use it.

$\rightarrow$ R s (SEXP s 'R.Expr)
parse file n txt = ⟦r| `parse(file_hs, n_hs, txt_hs)` ⟧

Now that we have a Haskell function for parsing R source files, with a Haskell type signature, the Haskell compiler can check that all calls to `parse()` are well-formed. We found this feature immensely useful to document in the source code itself how to call various R functions, without having to constantly look up this information in the R documentation.

Of course, while form indexing SEXP can in practice be a useful enough surrogate for a real type system, it does not replace a real type system. A reasonable property for any adequate type system is *type preservation*, also called *subject reduction*. That is, we ought to have that:

$$\text{If } \Gamma \vdash M : T \text{ and } M \Downarrow V \text{ then } \Gamma \vdash V : T$$

where M is an expression, T is a type and V is the value of M. The crude form of indexing presented here does not enjoy this property. In particular, given some arbitrary expression, in general the form of the value of this expression is unknown. We have the following type of SEXP's of unknown form:

data SomeSEXP $s = \forall a.$SomeSEXP (SEXP s a)

Because the form of a value is in general unknown, the type of eval is:

eval :: SEXP s $a \rightarrow$ R s (SomeSEXP s)

That is, for any SEXP of any form a, the result is a SEXP of some (unknown) form.

3.2.3 Casts and coercions

SEXP's of unknown form aren't terribly useful. For example, they cannot be passed as-is to the successor function on integers, defined as:

succ :: SEXP s 'R.Int $\rightarrow$ R s SomeSEXP
succ x = ⟦r| `x_hs + 1` ⟧

Therefore, H provides *casting functions*, which introduce a dynamic form check. The user is allowed to *coerce* the type in Haskell of a SEXP given that the dynamic check passes. cast is defined as:

cast :: SSEXPTYPE $a \rightarrow$ SomeSEXP $s \rightarrow$ SEXP s a
cast ty s
  | fromSing ty $\equiv$ R.typeOf s = unsafeCoerce s
  | otherwise = error `"cast: Dynamic type cast failed."`

where SSEXPTYPE is a singleton type reflecting at the type level the value of the first argument of cast. The use of a singleton type here allows us to to write a precise specification for cast: that the type of the return value is not just any type, but uniquely determined by the value of the first argument ty.

Now, ⟦r| `1 + 1` ⟧ stands for the *value* of the R expression "`1 + 1`". That is,

two = ⟦r| `1 + 1` ⟧ :: SomeSEXP s

In order to compute the successor of two, we need to cast the result:

three :: R s SomeSEXP
three = succ (two 'cast' R.Int)

3.3 R values are (usually) vectors

An idiosyncratic feature of R is that scalars and vectors are treated uniformly, and in fact *represented* uniformly. This means that provided an interface to manipulate vectors alone, we can handle all scalars as well as all vectors. H exports a library of vector manipulation routines, that mirrors the API of the standard `vector` package.

The advantage of keeping data represented as R vectors throughout a program is that no marshalling or unmarshalling costs need be incurred when passing the data to an R function. Because we provide the exact same API as for any other (non-R) vector representations, it is just as easy to manipulate R vectors instead, throughout.

3.4 Memory management

One tricky aspect of bridging two languages with automatic memory management such as R and Haskell is that we must be careful that the garbage collectors (GC) of both languages see eye-to-eye. The embedded R instance manages objects in its own heap, separate from the heap that the GHC runtime manages. However, objects from one heap can reference objects in the other heap and the other way around. This can make garbage collection unsafe because neither GC has a global view of the object graph, only a partial view corresponding to the objects in the heaps of each GC.

3.4.1 Memory protection

Fortunately, R provides a mechanism to "protect" objects from garbage collection until they are unprotected. We can use this mechanism to prevent R's GC from deallocating objects that are still referenced by at least one object in the Haskell heap.

One particular difficulty with protection is that one must not forget to unprotect objects that have been protected, in order to avoid memory leaks. H uses "regions" for pinning an object in memory and guaranteeing unprotection when the control flow exits a region.

3.4.2 Memory regions

There is currently one global region for R values, but in the future H will have support for multiple (nested) regions. A region is opened with the runRegion action, which creates a new region and executes the given action in the scope of that region. All allocation of R values during the course of the execution of the given action will happen within this new region. All such values will remain protected (i.e. pinned in memory) within the region. Once the action returns, all allocated R values are marked as deallocatable garbage all at once.

runRegion :: ($\forall s.$R s a) $\rightarrow$ IO a

Regions are transactions, in that protection prevails during the transaction and ceases after transaction closure.

3.4.3 Automatic memory management

Nested regions work well as a memory management discipline for simple scenarios when the lifetime of an object can easily be made to fit within nested scopes. For more complex scenarios, it is often much easier to let memory be managed completely automatically, though at the cost of some memory overhead and performance penalty. H provides a mechanism to attach finalizers to R values. This mechanism piggybacks Haskell's GC to notify R's GC when it is safe to deallocate a value.

automatic :: MonadR m $\Rightarrow$ R.SEXP s $a \rightarrow$ m (R.SEXP G a)

In this way, values may be deallocated far earlier than reaching the end of a region: As soon as Haskell's GC recognizes a value to no longer be reachable, and if the R GC agrees, the value is prone to be deallocated. Because automatic values have a lifetime independent of the scope of the current region, they are tagged with the global region G (a type synonym for GlobalRegion).

For example:

do $x \leftarrow$ ⟦r| `1:1000` ⟧
  $y \leftarrow$ ⟦r| `2` ⟧
  return \$ automatic ⟦r| `x_hs * y_hs` ⟧

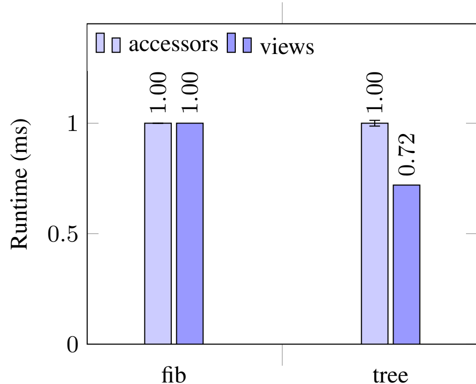

Figure 5. Normalized comparison of runtimes between views and accessor functions.

Automatic values can be mixed freely with other values.

4. Benchmarks

As explained in Section 3.1, there are essentially two ways to deconstruct a SEXP value returned from R: using accessor function provided by R's C extension API, or construct a view of the SEXP as an algebraic datatype and perform case analysis on that. The latter approach is more convenient, but potentially slower, because it nominally implies allocating a view data structure. However, as argued in Section 3.1, through careful engineering of the view, we expect the compiler to optimize away any extra allocation. In this section, we test this hypothesis experimentally.

To compare the performance cost of using a view function, we implement two micro benchmarks. The `fib` benchmark measures the performance of a naive implementation in Haskell of the Fibonacci series over R integers. Likewise, the `tree` benchmark concerns a binary tree traversal function implemented in Haskell, where the tree is represented using generic R vectors. The results[6] are provided in Figure 4. They were obtained using the Criterion benchmarking tool [10] using the default settings on a lightly loaded machine. Where the standard deviations are significant ($\not< 1\%$), we indicate them with error bars.

As expected, using a view does not significantly impact performance. In fact, as can be seen in the `tree` benchmark, views can even be faster than using accessor functions. The reason is that a view function can be completely inlined, yielding code that makes direct memory accesses to statically known offsets in memory, given a pointer to a SEXPREC structure. Accessors, on the other hand, are C functions that are opaque to the compiler — they cannot be inlined, meaning that the overhead of calling a C function, using the standard calling convention, must be incurred at each field access.

5. Related Work

TODO

6. Conclusion

Given modern extensions to the Haskell programming language, the language turns out to support surprisingly easy interoperability with other languages, with opt-in static guarantees on what arguments get passed to a function, but without the hassle of elaborate bindings. The enabling ingredient here is the existence of a quasiquotation mechanism, which makes it easy to express calls to foreign language functions in the foreign language itself, in a way that matches up the foreign documentation for that function. Many of the ideas in this paper can be transposed to any other functional language provided an equivalent quasiquotation mechanism exists (e.g. Camlp4 [8] or extension points in OCaml, or SML quotation [11]). In a multistage language but without quasiquotation, the strategy for avoiding marshalling costs would still apply, but constructing foreign calls would likely turn out rather less convenient.

Interacting with a latently typed language necessarily induces some number of runtime checks. While foreign functions can be selectively typed in the host language, this in practice requires a pervasive use of casts and coercions. Casts are runtime checks that have a cost, but moreover a cast failure is not very informative as to why it happened. A future direction could include integrating a notion of *blame* [3] into H, in order to better pinpoint the true cause of a dynamic check failure.

[6] Benchmarks available at `http://github.com/tweag/H`.

References

[1] M. M. Chakravarty. C ⟶ HASKELL, or yet another interfacing tool. In *Implementation of Functional Languages*, pages 131–148. Springer, 2000.

[2] M. M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. P. Jones, A. Reid, et al. The Haskell 98 foreign function interface 1.0. *An Addendum to the Haskell Report*.

[3] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN Notices*, volume 37, pages 48–59. ACM, 2002.

[4] GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.3.

[5] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[6] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *ACM Sigplan Notices*, volume 44, pages 1–12. ACM, 2008.

[7] G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[8] M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations for ml. In *Proc. 1994 Workshop on ML and its applications*, pages 70–78, 1994.

[9] R. Milner, M. Tofte, R. Harper, and D. MacQueen. The Definition of Standard ML, revised edition. *MIT Press*, 1(2):2–3, 1997.

[10] B. O'Sullivan. Criterion, 2014. URL `http://www.serpentine.com/criterion/`.

[11] K. Slind. Object language embedding in standard ml of new jersey. In *Proceedings of the Second ML Workshop, CMU SCS Technical Report. Carnegie Mellon University, Pittsburgh, Pennsylvania*, 1991.