HaPy
Haskell for Python

David Fisher, Ashwin Siripurapu, and William Rowan

December 17, 2011

1 Introduction

HaPy is an implementation of a foreign function interface from Python to Haskell with a strong focus on ease of use. In particular, HaPy entirely eliminates the need for FFI-related boilerplate code in either language. That is, Haskell modules can be used from Python without any modification. Moreover, from the programmer's perspective, imported Haskell functions are indistinguishable from native Python functions and Haskell modules are imported using the standard Python import syntax.

2 Motivation

Each programming language has its own unique set of strengths and weaknesses. Therefore, choice of language is important when starting a project. In a perfect world, we would be able to mix and match languages at will, using the perfect language for each part of a project. Unfortunately, interfacing different languages often causes significant overhead, more than eliminating the gains reaped from the language specialization. The authors have found that, in general, the usefulness of a foreign function interface is directly proportional to its ease of use. Therefore, we hope to entirely remove the overhead of calling Haskell from Python.

Three trends are clear when looking at the most recent State of Haskell survey[2] and other data: in the first place, Haskellers primarily use Haskell for mathematical analysis, parsing/compiling, and (increasingly) web development. Importantly, GUI development in Haskell is almost non-existent. Secondly, Python is a popular language for creating GUIs due to the existence of a variety of easy to use toolkits. Lastly, Haskell is much faster than Python. This suggests an example use-case: an application with a Python GUI and a Haskell backend.

3 Use

Our goal with HaPy is to allow the user to call arbitrary Haskell code while remaining committed to "Pythonic" syntax. In particular, importing a Haskell module into a Python program is easy with HaPy: simply add the line

```
from HaPy import HsModule
```

or

```
import HaPy.HsModule
```

to the head of any Python script. This will dynamically load the specified Haskell module, so that any Haskell function defined in HsModule can be accessed and executed by the Python script, as though it were code defined in another Python module. For instance, if HsModule defines a function `sum` which takes two real numbers and returns their sum, the result can be printed in Python with

```
print HsModule.sum(first, second)
```

Installing HaPy is similarly intuitive, but with the caveat that all libraries called by `HaPy.hs` must be dynamically-linkable. This means that all of the packages that `HaPy.hs` imports must be reinstalled with Cabal, using the `--enable-shared` flag. [1]
However, Haskell packages that are used by the user's Haskell code need not be dynamically-linkable. Indeed, on the Haskell side, no special support is needed by the user; any standard Haskell code will run with HaPy (modulo the incompleteness of our type checker). All the issues of marshalling data and passing data to and from the Haskell foreign function interface are managed by the code in `HaPy.hs`. This has the added benefit of enabling the user to call any code defined in the Haskell standard libraries from Python.

3.1 Loading Haskell Modules

As noted previously, the syntax for loading Haskell modules is identical to the normal Python import syntax. All Haskell modules are represented as submodules of the HaPy module, so that the Python interpreter should not mistake them for Python modules.
Furthermore, the Haskell module hierarchy is preserved, so for instance one can say

```
import HaPy.Data.List
```

3.2 Haskell Types in Python

One of the main challenges in HaPy is the representation of Haskell data types in Python. Some primitive Haskell types (currently `Bool`, `Int`, `Double`, and `String`) are marshalled into the equivalent Python type. All other Haskell types are returned as opaque references: they can be passed to Haskell functions, but cannot be modified or inspected by other Python code. Use of Haskell functions is trivial: they can be called just like other Python functions. Arguments can be any of the supported primitive types, which will be marshalled, or Haskell object references, which are passed through. The return value will be automatically converted into a Python type if possible; otherwise, it will remain as an opaque Haskell reference. The Haskell functions do not behave exactly like other Python functions: they support currying. When they are called with too few arguments, a partially-applied function object is returned. This does not invalidate the original function - all functions and partially-applied functions can be called as many times as desired. We chose to do this because it naturally followed from the implementation (see **??**) and nicely mirrors the behavior of actual Haskell functions. Further, this functionality can be ignored if not desired. (The user only loses immediate too-few-argument errors.)

[1] The requirement that these packages be dynamically-linkable stems from the fact that `ctypes` must be able to dynamically link to `HaPy.hs` (see **??**)

4 Implementation

4.1 Dynamic Loading of Haskell Modules

HaPy uses the HSPlugins package to dynamically load requested Haskell modules. HSPlugins supports more than just dynamic loading of external object files. Most of the meat of the package is in two crucial components: the loading of dependent modules and dynamic type checking.

HaPy doesn't actually use the dynamic type checking features of HSPlugins. This feature requires that client Haskell supply static types to check against. This make sense for HSPlugins intended use case, the loading of plugins that implement known interfaces. We, however, need to be able to load any Haskell module with any interface at runtime. We therefore ask HSPlugins to load all symbols as an opaque data type (literally `data Opaque`) which will loose its type information when passed back to the Python side anyway.

HSPlugins supports a simple, but limited interface for loading modules. The client is expected to pass a string representing the path to the object file containing the requested module and a string representing the symbol to be loaded. An implicit expectation is that the Haskell interface file for the requested module sits next to the object file on the file system and has the same base name.

While this condition holds for locally compiled modules (GHC by default generates the object file and interface file next to each other) this doesn't hold for Haskell packages installed on the system. Loading modules from these packages posed several challenges. The first is finding the location of a package's object file from the module name. The module `System.Plugins`, for example, might be provided by the package `plugins-1.5.1.4`. Fortunately, we are able to use the GHC API to do this lookup for us which we then translate into a path to the object file.

Our method of doing so is not robust, however. Ideally, we would be able to ask GHC to invoke the same mechanism that it uses to lookup modules when the user requests packages in GHCi but the complexity and opacity of the GHC API made it impossible to do this in our timeframe.

The second challenge involved satisfying HSPlugins assumption that the Haskell interface file sits next to the object file for the module. Packages might provide many different modules, the interface files for which are stored underneath the the package's base directory in a file structure mimicking the package structure. We had to thus copy the requested interface file to the location expected by HSPlugins.

Again, this mechanism is not robust. It will be important to utilize GHC's import mechanism in the future. Properly importing both local and package modules will probably require either utilizing HSPlugins' low level API or abandoning it altogether.

4.2 Function Currying

The Python-side wrapper around Haskell functions is implemented as a variable-argument callable Python object, which contains type information and an opaque pointer to the Haskell function. To evaluate an opaque Haskell function pointer from Python, it must be passed to a conversion function in Haskell (using the FFI) with its arguments, where it is re-cast as the appropriate type, applied, and returned to Python. Instead of many different conversion functions for different numbers of arguments, there is a single-argument conversion function for each ar-

gument type (one for each of the supported marshalled types and one for opaque types). Therefore, function calls to Haskell function objects are curried in Python, and arguments are pushed through the FFI one at a time. When there are no arguments left (as determined by the type checker), Python converts the opaque pointer to the known return type of the Haskell function.

4.3 Type Checking

As Haskell is statically typed, no runtime type checks occur when calling Haskell functions. If a dynamically loaded Haskell function is called with the wrong type, silent memory corruption occurs. This generally leads to a segfault, but can also lead to other sorts of nastiness (incorrect return values, etc). While this only occurs in the case of programmer error (i.e. passing incorrect arguments to a function), the result is not acceptable: a reasonable type error should be generated. Therefore, dynamic type checking is essential. There are two possible approaches to dynamic type checking. The first - and more preferable - approach is to use GHC's type checker. The advantage of this approach is clear: types will (almost tautologically) always be checked correctly, as - for our purposes - GHC is the canonical implementation of type checking. The second approach is to do anything else, which more or less boils down to writing your own type checker. Unfortunately, a full copy of GHC's type checker would be fairly difficult to write, so this leaves the type checker with some subset of types that it can properly check (the size of which is about logarithmically related to the work involved). We started with the first approach, but ran into major difficulties with the internal GHC API. It is largely undocumented and is complicated enough to make it very difficult to understand. On top of this, it appears fairly susceptible to change: we noticed some differences between GHC 7.0.3 (which we had installed) and GHC 7.2.2 (the latest release at this time). For these reasons, we decided to use the second approach. The current type checking system extracts the type information from Haskell files using GHCi's `:browse` command. This provides information for both installed packages and local files. (While it does not require the Haskell source for installed packages, through some quirk it will not give type information for local files unless the source is present. However, we find this acceptable for the moment.) Because we are calling GHCi externally, we receive this type information as a string. While we could have done this from any of our three project languages, we chose to keep track of the type information in Python. This lets the Python code know how to marshal specific data types before and after function calls and when it should request the final result of a function (i.e. when all the arguments have been applied) without adding additional signalling complexity to the FFI. Furthermore, Python can be fully in charge of displaying type errors. Currently, the type checker does not support a large class of different function types. Type classes are ignored. Any complex polymorphism does not work. Any data types that are fully specified in the type signature work.

5 Future Work

While we have succeeded in making HaPy easy to use, HaPy is not nearly robust enough to use in a production application as of yet.

5.1 Type Checking

The current type checker is fragile for a couple reasons. First, it parses an output that is intended to be human readable and therefore lacks a formal specification and is subject to change. Second, most of the type checking is string-manipulation based, which is fragile. In the future, we would like to use GHC's type system. Although the API is difficult to understand, it is robust. Furthermore, there's no other way to get type checking completely right.

5.2 Supporting More Types

In addition to marshalling primitive types across the Haskell-Python interface, we would eventually like to be able to marshal Python lists into Haskell lists and vice versa. This would almost certainly be done through a C wrapper function for efficiency. One potential challenge here would be the marshalling of infinitely long, lazily-evaluated lists from Haskell to Python.
Another challenge would be dynamically converting from algebraic data types (ADTs) defined in Haskell to Python objects; for example, a simple product type composed of Haskell primitives could be represented as a named tuple[2] in Python, provided that it is created using record syntax. Surprisingly, non-record ADTs will be much harder to translate into Python types because the only way (to the authors' knowledge) of extracting their values is pattern matching.

5.3 Improved Dynamic Loading

HaPy uses `System.Plugins` to dynamically load Haskell modules, which assumes that the module's `*.hi` file is located in the same directory as the corresponding `*.o` file. As previously mentioned, this is not the case for packages. This forces us to do a kludgey workaround: we temporarily copy the `*.hi` file into the same directory `*.o` file before loading them. In the future, we would either like to move away from `System.Plugins` or switch to its lower level API so we do not have this problem.

5.4 Simplified Install

While developing HaPy, we encountered a serious bug relating to dynamic linking. In 64-bit environments, this results segfaults when using the Haskell `System.Plugins` library.
Further, the Python `ctypes` module dynamically links to our binary. Unfortunately, this requires that we build our binary with the `-dynamic` flag, which causes GHC to link all of the libraries HaPy uses dynamically. This means that users of HaPy must use Cabal to reinstall several packages with the `--enable-shared` option. We should be able to statically link all of our libraries, so that we can simply distribute a self-contained binary. Furthermore, we speculate that this might solve our 64-bit bug.

5.5 Passing Python Callbacks

Lastly, we would like to allow users to pass Python functions to Haskell; an example use case might be passing a comparison function for Haskell to use in sorting a list. Python's `ctypes` module already supports passing Python functions to C code (provided that the functions take only arguments which can be converted to C types), so presumably we would use this to facilitate the passing of Python function pointers to Haskell. Once we have a pointer to a Python function in C, Haskell data can be marshalled

[2] Effectively a fixed-size dictionary

into an appropriate `ctype` for the function to act on, and the return type of the function can subsequently be converted back into a Haskell type.

References

[1] Stewart, Don, *Haskell Plugins*. `http://hackage.haskell.org/package/plugins`

[2] Tibell, Johan, *State of Haskell Survey, 2011*. `http://blog.johantibell.com/2011/08/results-from-state-of-haskell-2011.html`