Foreign Inline Code

Systems Demonstration

Manuel M. T. Chakravarty
University of New South Wales, Australia
chak@cse.unsw.edu.au

Extended Abstract

Template Haskell, the Glasgow Haskell Compiler's (GHC) meta programming framework [4], is widely to used define macros, code generators, or even code transformation engines. Mainland [2] recently extended Template Haskell with support for quasiquoting arbitrary programming languages, which greatly simplifies writing code generators that produce complex C, CUDA, OpenCL, or Objective-C code by writing code templates in the syntax of the generated language—for example, *Accelerate*, an embedded language for GPU programming, makes extensive use of that facility to generate CUDA GPU code [3].

In this demo, I will show that quasiquoting also enables a new form of language interoperability. Here, a simple example using Objective-C:

```
nslog :: String -> IO ()
nslog msg =
  $(objc
      ['msg :> ''String]
      (void
        [cexp| NSLog(@"A message from Haskell: %@",
                         msg) |]))
```

The expression splice `$(objc ...)` introduces an *inline* Objective-C expression into Haskell code. It's first argument (which here is `['msg :> ''String]`) is a list of all Haskell variables used and automatically marshalled to Objective-C code. The syntax `'msg` is Template Haskell to quote a variable name and `''String` to quote a type constructor name. The infix operator `(:>)` is used to annotate variables with marshalling information, in this case, the type used for type-guided marshalling.

The quasiquoter `[cexp|...|]` quotes C expressions, returning a representation of the quoted expression as an abstract syntax tree. Here, the expression calls the function `NSLog()`, which on OS X and iOS writes a log message.

As Objective-C is a strict superset of ANSI C, this works for inline ANSI C code as well. With appropriate support by a quasiquotation library, this approach could also be used for other languages, such as Java or C++. It might even be plausible to inline scripting languages, such as Ruby or Python.

Haskell '14, September 4–5, 2014, Gothenburg, Sweden.

The trouble with bridging libraries. What is the benefit of using foreign code inline? An important design goal of Haskell 2010's *foreign function interface* was simplicity. It was designed to be sufficiently versatile to support all major use cases, but still be simple, so that it can be easily supported by Haskell implementors. In addition, the intention was that more powerful tools would be built on top of it — tools like `hsc2hs` and C→Haskell (`c2hs`) [1]. Usually, these tools are then used to write *bridging*, *binding*, or *wrapper* libraries for existing foreign libraries; an example of such a library is the `gtk` package.

Unfortunately, bridging libraries suffer from a limitation of scale. Modern platform libraries, such as those for Android, iOS, OS X, and Windows are huge; so, writing comprehensive bridging libraries would be an enormous task. Even worse, due to the rapid evolution of those platforms, the maintenance of bridging libraries would require considerable resources.

Moreover, while Haskell's FFI support for C is comprehensive, support for interacting with object-oriented languages, such as C++, C#, Java, and Objective-C, is barely existent. This is despite a number of attempts to support those languages more directly.

The advantage of foreign inline code. Foreign inline code eliminates the need for bridging libraries, as foreign libraries can simply be accessed in their native language in the middle of a Haskell module. The `language-c-inline` library takes care of putting all foreign code into a separate file, generating all marshalling code and foreign declarations as well as inserting all inter-language calls.

With this approach, the use of foreign libraries is usually more coarse-grained: instead of calling a foreign function at a time from

Haskell, foreign inline code often combines multiple foreign library calls into one FFI invocation in an application-specific manner. This potentially lowers the overheads of cross-language calls in an application and effectively uses an application-specific cross-section of a foreign library.

Somewhat surprisingly, inline code requires a less tight coupling of a foreign language with Haskell than bridging libraries. All previous attempts to support libraries that make heavy use of subclassing and inheritance have tried to model these mechanisms in Haskell using a wide array of type-level trickery. In contrast, foreign inline code can use these mechanisms in the foreign language itself, without attempting to encode these mechanisms in Haskell. This includes the common case of subclassing in GUI frameworks, which we support by allowing inline code at the toplevel as declaration forms using toplevel Template Haskell splices, not just embedded in expressions. Overall, inline code appears to significantly simplify using libraries written in object-oriented languages, such as C++, C#, Java or Objective-C, from Haskell.

A reasonable objection to inline code is that a developer needs to be fluent in two languages, instead of just one. However, the limited abstraction provided by medium to large bridging libraries usually requires being familiar with the native API and relying

on native documentation as well. For example, to use Haskell's `OpenGL` package, you need a detailed understanding of the OpenGL C API.

The demo. Beginning with the example quoted above, the demo will illustrate `language-c-inline` at a few examples of increasing complexity. The most involved is a simple GUI application that provides a Haskell REPL in a window. It will make use of Apple's Cocoa framework, which would be a huge undertaking to support with a bridging library. By way of these examples, I will outline how to use `language-c-inline` and what kind of code it produces.

The topics covered include the following:

- Foreign inline code in expressions.
- Using Haskell variables and functions in foreign inline code.
- Marshalling data between Haskell and the inlined language.
- Inline definitions of new (sub)classes.
- Semi-automatic generation of proxy classes for Haskell structures.

The library `language-c-inline` is available from Hackage. Its GitHub repository is at

`https://github.com/mchakravarty/language-c-inline`

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classification—Applicative (functional) languages; Object-oriented languages; D.3.4 [*Programming Languages*]: Processors—Code generation

Keywords Interoperability; Inline code; Template meta-programming

References

[1] M. M. T. Chakravarty. C→Haskell, or yet another interfacing tool. In P. Koopman and C. Clack, editors, *Implementation of Functional Languages, 11th International Workshop (IFL'99), Selected Papers*, number 1868 in LNCS. Springer-Verlag, 2000.

[2] G. Mainland. Why it's nice to be quoted. In *Haskell Symposium*, page 73, New York, New York, USA, 2007. ACM Press.

[3] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP: International Conference on Functional Programming*, Sept. 2013.

[4] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on*

Haskell, pages 1–16. ACM, 2002.