Foreign Calls in Haskell

N. Ramsey and K. Cronburg, for COMP 250RTS

November 27, 2017

Recommended reading

When Simon Peyton Jones works on a hard problem, he tends to publish solutions in stages—his style is a number of small, incremental papers rather than one definitive paper. To understand how foreign calls work in Haskell, then, we'll need to integrate material from several papers.

- *H/Direct: A Binary Foreign Language Interface for Haskell* provides the best overview:
  - Read sections 1 and 2. They describe the problem and the major design alternatives, with an overview of an IDL-based solution. Not all the details in section 2.3 are important; focus on the two levels of Haskell types and how they relate to the IDL types.
  - Skip section 3. The concepts are important, but I think I can explain them in five minutes in class more easily than you can get them from the section. If you want to build something like H/Direct, section 3 would be valuable, but if we just want to understand how it works, there are better ways.
  - In section 4, read about the five kinds of pointer in the bullets on page 157.
  - The concept of marshalling is essential, but the details given in this paper are not. If you can grasp the idea behind the example function `marshalPoint`, which appears at the bottom right of page 159, then you have everything you need from sections 5 and 6.
  - For our purposes, the later sections are superfluous, but you might want to note the claim in the second paragraph of section 9 on page 162:

    > We do not claim great originality for these observations. What is new in this paper is a much more precise description of the mapping between Haskell and IDL than is usually given. This precision has exposed details of the mapping that would otherwise quite likely have been mis-implemented. Indeed, the specification of how pointers are translated exposed a bug in our current implementation of H/Direct. It also allows us automatically to support nested structures and other relatively complicated types, without great difficulty. These aspects often go un-implemented in other foreign-language interfaces.

    Another win for formal semantics!

- Some of the key underlying technology is described only in *Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell*. We're recommending sections 4, 5.6, and possibly 6.
  - Read section 4 and understand "stable names" and "stable pointers." Section 4.4 is especially apropos; to understand some implications for garbage collection, read it closely.
  - Read section 5.6, just so you understand the memory-managment issue—not necessarily the solution.
  - Section 6 describes finalization. If you're not already familiar with the concept, read the opening of this section (four paragraphs). The rest of the section is a little obscured by details of weak pointers, but the key points are there.

    If I wanted to study finalization, I would study the brief review of the literature in section 8. For an idea of the range and complexity of the design space, I might continue with the survey paper by Barry Hayes. I would also pay close attention to Dybvig's work on *guardians*, which has been recommended to me by people I trust.

- The H/Direct paper describes marshalling and unmarshalling. *Calling Hell From Heaven and Heaven From Hell* completes the picture. The paper is (regrettably) obsessed with COM objects, which were a common obsession in the late 1990s, before the JVMs and the browsers took over.
  - The key item in the introduction is the second bullet. (The first bullet recapitulates H/Direct, and the remaining bullets address arcana of COM.)
  - We care only about section 3. Pay special attention to sections 3.3 and 3.5 (stable pointers, higher-order callbacks).

- In *Programming in Lua* (Lua 5.0 edition), read section 27.3.1 on the *registry*.

Bonus reading

Hayes (1992), "Finalization in the collector interface," describes the (chaotic) state of finalization in the early 1990s.

Dybvig, Bruggeman, and Eby (1993), "Guardians in a generation-based garbage collector," describe *guardians*, which respected colleagues tell me is still the best of the finalization ideas.

Questions

(1) Analyze the type system of the IDL described in the H/Direct paper (figures 2 and 5), and describe how you would specify *Lua/Direct*, an IDL compiler for Lua.

(2) Page 117 of "Calling Hell..." describes a function `freeHaskellFunctionPointer`.

   (a) What methodology would you recommend for figuring out when to call this function?

   (b) Suppose Lua is playing the role of Haskell and is calling into and out of binary codes with a COM interface. Would it be appropriate for the Lua system to provide an analog of this function? Why or why not?

(3) Section 3.5 of "Calling Hell..." shows an example that is relentlessly higher-order: Haskell calls C calls Haskell. Let's consider the same example done in Lua.

   (a) What issues of having values cross the inter-language boundary are the same in both Lua and Haskell? What are different? Why?

   (b) In particular, how should Lua handle the problem of sharing a heap-allocated value with foreign code, given that the heap-allocated value might move?

   GHC uses a *stable pointer*. Does Lua have something analogous? If so, what is it? If not, how would you simulate a stable pointer? Or would you recommend a different solution entirely?

   (Note: Lua has "weak tables," with a complicated semantics that is described informally in the reference manual. Lua also has a form of finalizer.)