A Foreign Function Interface for Haskell

John Peterson

Research Report YALEU/DCS/RR-971

Yale University

Department of Computer Science

New Haven, CT 06520

1 Introduction

Recent work has shown how purely functional languages such as Haskell can be integrated with imperative languages by using *monads* to encapsulate the state of the imperative computation. This preserves the referential transparancy of the functional language and avoids placing conffffffffffffffffffffffffffffffffffffffffffffffff

This annotation, `LispName`, takes a single argument: a string containing the Lisp object to be used as the `add` function. Since in Lisp, `+` is used for addition, the following interface would work:

```
interface Add where

add :: Integer -> Integer -> Integer
{-# add :: LispName("+") #-}
```

This indicates to the Haskell code generator that calls to the Haskell `add` function should be translated to calls to the Lisp `+`. The following program uses `add`:

```
module Main where

import Add

main = print (add 1 2)
```

A number of things happen behind the scenes for this example to work. First, there is an implicit conversion of the datatype `Int` to its equivalent Lisp representation. Second, there is an implicit conversion of the function calling protocols between the Lisp and Haskell worlds — Haskell's call protocol supports currying and laziness, while the Lisp protocol does not. When the interface is compiled a *wrapper* function is generated which takes care of the call conversion. By default, the system determines the arity of the function from its type signature and passes the correct number of arguments to the Lisp function. Also by default, the Lisp function is assumed to be strict: all parameters are evaluated before being passed to the Lisp code.

In this particular example, the default behavior makes everything work as it should. In more complicated situations, you must provide some additional information explicitly; this is described in detail in the next section.

This same example can also be done using a C implementation of `add`. The file `add.c` contains

```
int add(x,y)
  int x,y;
{return (x+y);
};
```

The interface would be

```
interface Add where

add :: C_int -> C_int -> C_int
{-# add :: CName("add") #-}
```

The type `C_int` is a synonym for `Integer` used in interface files to select the exact numeric representation used in the C code. All C numeric types are represented by `Integer` except `float` and `double`, which have direct Haskell equivilants. Coercion between C and Haskell numerics is automatic.

In most cases, you need to load the file (or files) containing the foreign functions being called. This is accomplished by denoting `.lisp`, `.scm`, or `.o` files in the `.hu` file defining the compilation unit. The `.lisp` extension is used for Common Lisp files; the `.scm` extension is used for the Scheme dialect used in the Yale Haskell implementation. Either type of lisp file will be compiled when the associated compilation unit is compiled. C files must be compiled separately; their compilation is not managed by the Haskell system. Use `cc -c file.c` to compile C files used by Haskell programs.

A file containing Lisp code can be loaded into any Common Lisp package. Lisp objects referenced in annotations are, by default, in the `MUMBLE-USER` package. This package is used for the implementation of the compiler and code in this package is in a Scheme-like dialect of Lisp. Real Common