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 conffffff

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

Lisp code should be placed in a different package, usually the `user` package. Using a package prefix in the `LispName` annotation ensures that the the correct Lisp function is referenced. Finally, the Lisp definition need not be a function: a macro may also be used.

The following example elaborates on the previous lisp based addition example. One way of greatly increasing the performance of the Lisp code is to add type declarations. With appropriate declarations, the file of Lisp code might look like this:

```
(in-package 'user)

(defun add (x y) (the fixnum (+ (the fixnum x) (the fixnum y))))
```

The interface file might be done like this:

```
interface Add where

add :: Int -> Int -> Int
{-# add :: LispName("user::add") #-}
```

The `Int` type in Haskell is implemented by the `fixnum` type in the underlying lisp.

3 Data Conversions

In the previous example, the Lisp and Haskell programs shared a common representation of the data types used. In the case of more complex structures, conversions are required to transform lazy Haskell structures into the their non-lazy equivalants and perform other changes in representation. Yale Haskell does this automatically for the following data types: `Bool`, `Int`, `Integer`, `Float`, `Double`, lists, `Char`, and `String`. The exact type vocabulary for each language varies and is described later.

For example, an interface such as

```
interface Concat where

stringConcat :: String -> String -> String

-- assume string-concat is defined in the Lisp world
{-# stringConcat :: LispName("user::string-concat") #-}
```

works as expected even though Haskell and Lisp use different string representations.

Occasionally, the automatic conversions for Lisp structures are not needed. Conversion to

system views the object as a reference which names part of the global state. Thus the object itself remains referentially transparent even though it contains mutable structure.

A good example of this is a Lisp hash table. The following interface defines a simple hash table library using strings as both the keys and values in the hash table.

```
interface Hash where

data HashTable  -- an abstract data type

newTable :: IO HashTable
getTable :: HashTable -> String -> IO String
putTable :: HashTable -> String -> String -> IO ()
{-# newTable :: LispName("user::new-hash-table")
    getTable :: LispName("user::get-hash-table")
    putTable :: LispName("user::put-hash-table") #-}
```

The Lisp code would look like this:

```
(in-package 'user)

(defun new-hash-table () (make-hash-table :test #'equal)

(defun get-hash-table (table key) (gethash key table ""))

(defun upate-hash-table (table key value) (setf (gethash key table) value))
```

Objects of type `HashTable` can be used freely in the Haskell program. However, only calls to the monad via `readTable` and `putTable` can be used to reveal the mutable component of the table. Even though `readTable` itself has no side effect, it must be declared using the `IO` type since the value it returns depends on the result of changes to the program state.

6 Sharing Data Structures

The C interface does not yet support shared data structures.

Abstract data types allow only a limited form of sharing between Lisp and Haskell. An abstract data type does not allow direct access to the internal structure of an object through pattern matching or construction. You can accomplish a more general sharing of objects by defining an object in terms of a set of functions which construct new values and select components from existing values of that type.

An operational view of a data constructor is simply a set of functions which test for, build, and select fields from a value. Thus a constructor of arity k is represented by $k + 2$ functions: a test predicate, a constructor function that accepts k arguments, and k component selectors. Data types with only one constructor do not need a test predicate; this is only necessary when more than one constructor is associated with a data type.

You can define the basic operations on a data type in either the Lisp or the Haskell world. You can import a data type defined in Lisp into Haskell using the `ImportLispType` annotation. This annotation can be placed either in a Haskell interface file or in an ordinary Haskell source code file. In either case, the annotation must be in the same module which defines the data type.

An `ImportLispType` annotation defines data types by giving the appropriate Lisp functions which operate on the type. For each constructor in the data type, you need to list the functions that implement the $k+2$ functions. The functions are ordered: the first is the test predicate, then the constructor function, and then the k selectors.

As an example, Haskell lists could are implemented using the following type declaration:

```
data List a = Null | Cons a (List a)

{-# ImportLispType(List(Null("lisp:null","'()"),
                        Cons("lisp:consp","lisp:cons","lisp:car","lisp:cdr")))
     #-}
```

The Lisp expressions in quotes need not be symbols: constants such as `'()` and even anonymous function definitions can be used. Note specifically that a nullary constructor must be a Lisp value, rather than the name of a function that is called with no arguments.

This definition of a list retains the laziness of Haskell. That is, the components of the list must be delays. The Haskell compiler inserts delays into any list created by `cons` in the above declaration and expects that `car` and `cdr` return delays. Any use of this structure from Lisp must use delays in the same manner.

Lazy structures can be avoided using strictness annotations. If the definition of the list type had been

```
data List a {-# STRICT #-} = Null | Cons a (List a)
```

then the elements of the structure would not be delayed. This declaration could be used to allow Haskell to directly make use of ordinary Lisp lists.

The next example elaborates on hash tables. We make a number of improvements in the previous hash table example. First, the tables are polymorphic in their data values. We still use strings as hash table keys but the values stored in the hash tables are unrestricted. Second, we make hash table construction non-strict in the data value. Implementationally, the values associated with the keys in the Lisp hash table are delay objects which are forced only when they are extracted from the table. Finally, we introduce a shared data type to indicate lookup failure. We implement this as `nil` for unsuccessful lookup. Since the values in the hash table are delay objects, the use of `nil` cannot be confused with actual data values. We also add a printer for hash tables.

```
interface HashPrims where

import HashTable(Maybe,HashTable)

newTable :: IO (HashTable a)
getTable :: HashTable a -> String -> IO (Maybe a)
putTable :: HashTable a -> String -> a -> IO ()

{-# newTable :: LispName("user::new-hash-table")
    getTable :: LispName("user::get-hash-table")
    putTable :: LispName("user::put-hash-table"),Strictness("S,S,N")  #-}

module HashTable(Maybe(..),HashTable,newTable,getTable,putTable) where
```

```
import HashPrims

data Maybe a = None | Just a deriving Text
{-# ImportLispType(Maybe(None("lisp:null","'()"),
                         Just("user::not-null",
                              "lisp:identity","lisp:identity"))) #-}

data HashTable a = HashTable a

instance Text(HashTable a) where
  showsPrec x p = showString "<Hashtable>"

(in-package 'user)

(defun new-hash-table () (make-hash-table :test #'equal))

(defun get-hash-table (table key) (gethash key table '()))

(defun put-hash-table (table key value) (setf (gethash key table) value))

(defun not-null (x) (not (null x)))
```

There are a number of subtle points to this example. By declaring the data structures in an ordinary Haskell module `Text` instances can be attached. This would not be possible if the declaration were placed in the interface file. Since Haskell does not allow abstract data types to be declared directly, the `Hashtable` type is declared with a constructor but this constructor is not exported. The strictness signature for `putHash` prevents evaluation of the value argument. Instead, this argument is passed as a delay object. The delay is passed out into the `Maybe` structure. Since this structure was not declared strict, the delay extracted from the hashtable becomes the delay within the `Maybe` structure. The representation of `Maybe` only works because a delay can never be `nil`, allowing proper discrimination between the `None` and `Just` constructor.

Analagous to `ImportLispType` is `ExportLispType`. This annotation causes the Haskell code generator at create the test, construction, and selection functions itself. These functions can then be called from Lisp code. Again, watch out because nullary constructors are defined as variables rather than functions of zero arguments.

7 A Summary of the Lisp Interface

The Lisp interface uses the annotation `LispName` to associate a Lisp function with a Haskell name.

Either the `.lisp` (Common Lisp) or `.scm` files may appear in .hu files. The Haskell system compiles lisp code as necessary.

The following data types are translated automatically: `Bool`, `Char`, `String`, lists, `Int`, `Integer`, `Float`, and `()`. Haskell data types are passed using whatever internal representation is used by the system. Abstract data types defined in the interface can be used to represent arbitrary Lisp objects.

The `NoConversion` annotation prevents structure translation.

A `Strictness` annotation can be used to pass unevaluated parameters.

Data structures can be shared between Haskell and Lisp using `ImportLispType` or `ExportLispType`.

8 A Summary of the C Interface

The C interface uses the annotation `CName` to associate a Lisp function with a Haskell name.

The `.o` extension in Unit (`.hu`) files load compiled C code. All C code must be compiled using `cc -c` before being loaded by Haskell.

The following data types are translated automatically: `Bool` represented by `C_bool`, `Char` is represented by `C_char`, `String` is represented by `C_string`, and `()` denotes the void type. The following numeric types are represented by `Integer` in Haskell: `C_short`, `C_int`, `C_long`, `C_unsigned_char`, `C_unsigned_short`, `C_unsigned_int`, and `C_unsigned_long`. The type `C_float` is represented by `Float` and the type `C_double` is represented by `Double`. All types in the signature of a C function must have the `C_` prefix - abstract types and other Haskell types are not yet supported.