H/Direct: A Binary Foreign Language Interface for Haskell

Sigbjorn Finne
University of Glasgow
sof@dcs.gla.ac.uk

Daan Leijen
University of Utrecht
daan@cs.uu.nl

Erik Meijer
University of Utrecht
erik@cs.uu.nl

Simon Peyton Jones
University of Glasgow
simonpj@dcs.gla.ac.uk

Abstract

H/Direct *is a foreign-language interface for the purely functional language Haskell. Rather than rely on host-language type signatures,* H/Direct *compiles Interface Definition Language (IDL) to Haskell stub code that marshals data across the interface. This approach allows Haskell to call both C and COM, and allows a Haskell component to be wrapped in a C or COM interface. IDL is a complex language and language mappings for IDL are usually described informally. In contrast, we provide a relatively formal and precise definition of the mapping between Haskell and IDL.*

1 Introduction

A foreign-language interface provides a way for programs written in one language to call, or be called by, programs written in another. Programming languages that do not supply a foreign-language interface die a slow, lingering death — good languages die more slowly than bad ones, but they all die in the end.

In this paper we describe a new foreign-language for the functional programming language Haskell. In contrast to earlier foreign-language interfaces for Haskell, such as Green Card [5], we describe a design based on a standard Interface Definition Language (IDL). We discuss the reasons for this decision in Section 2.

Our interface provides direct access to libraries written in C (or any other language using C's calling convention), and makes it possible to write Haskell procedures that can be called from C. The same tool also makes it allows us to call COM components directly from Haskell [4], or to seal up Haskell programs as a COM component. (COM is Microsoft's component object model; it offers a language-independent interface standard between software components. The interfaces of these components are written in IDL.)

H/Direct generates Haskell stub code from IDL interface descriptions. It is carefully designed to be independent of the particular Haskell implementation. To maintain this independence, *H/Direct* requires the implementation to support a primitive foreign-language interface mechanism, expressed using a (non-standard) Haskell `foreign` declaration; *H/Direct* provides the means to leverage that primitive facility into the full glory of IDL.

Because they cater for a variety of languages, foreign-language interfaces tend to become rich, complex, incomplete, and described only by example. The main contribution of this paper is to provide (part of) a formal description of the interface. This precision encompasses not only the programmer's-eye view of the interface, but also its implementation. The bulk of the paper is taken up with this description.

2 Background

The basic way in which almost any foreign-language interface works is this. The signature of each foreign-language procedure is expressed in some formal notation. From this signature, stub code is generated that *marshals* the parameters "across the border" between the two languages, calls the procedure using the foreign language's calling convention, and then unmarshals the results back across the border. Dealing with the different calling conventions of the two languages is usually the easy bit. The complications come in the parameter marshaling, which transforms data values built by one language into a form that is comprehensible to the other.

A major design decision is the choice of notation in which to describe the signatures of the procedures that are to be called across the interface. There are three main possibilities:

- *Use the host language (Haskell, in our case).* That is, write a Haskell type signature for the foreign function, and generate the stub code from it. Green Card uses this approach [5], as does J/Direct [8] (Microsoft's foreign-language interface for Java).
- *Use the foreign language (say C).* In this case the stub code must be generated from the C prototype for the procedure. SWIG [1] uses this approach.
- *Use a separate Interface Definition Language (IDL),* designed specifically for the purpose.

We discuss the first two possibilities in Section 2.1 and the third in Section 2.2.

2.1 Using the host or foreign language

At first sight the first two options look much more convenient than the third, because the caller is written in one language and the callee in the other, so the interface is conveniently expressed for at least one of them. Here, for example, is how J/Direct allows Java to make foreign-language calls:

```
class ShowMsgBox {
  public static void main(String args[])
  {
    MessageBox(0,"Hello!","Java Messagebox",0);
  }

  /** @dll.import("USER32") */
  private static native
    int MessageBox( int hwndOwner, String text
                  , String title, int fuStyle
                  );
}
```

The `dll.import` directive tells the compiler that the Java `MessageBox` method will link to the native Windows `USER32.DLL`. The parameter marshaling (for example of the strings) is generated based on the Java type signature for `MessageBox`.

The fatal flaw is that *it is invariably impossible, in general, to generate adequate stub code based solely on the type signature of a procedure in one language or the other.* There are three kinds of difficulties.

1. First, some practically-important languages, notably C, have a type system that is too weak to express the necessary distinctions. For example:
   - The stub code generator must know the mode of each parameter — `in`, `in out`, or `out` — because each mode demands different marshaling code.
   - Some pointers have a significant `NULL` value while others do not. Some pointers point to values that can (and sometimes should) be copied across the border, while others refer to mutable locations whose contents must not be copied.
   - There may be important inter-relationships between the parameters. For example, one parameter might point to an array of values, while another gives the number of elements in the array. The marshaling code needs to know about such dependencies.
2. On the other hand, it may not even be enough to give the signature in a language with an expressive type system, such as Haskell. The trouble is that the type signature still says too little about the foreign procedures type signature. For example, is the result of a Haskell procedure returned as the result of the foreign procedure, or via an out- parameter of that procedure? In the case of J/Direct, when a record is passed as an argument, Java's type signature is not enough to specify the layout of the record because Java does not specify the layout of the fields of an object and the garbage collector can move the object around in memory.
3. The signature of a foreign procedure may say too little about allocation responsibilities. For example, if the caller passes a data structure to the callee (such as a string), can the latter assume that the structure will still be available after the call? Does the caller or callee allocate space to hold the results?

In an earlier paper we described Green Card, whose basic approach was to use Haskell as the language in which to give the type signatures for foreign procedures [5]. To deal with the issues described above we provided ways of augmenting the Haskell type signature to allow the programmer to "customise" the stub code that would be generated. However, Green Card grew larger and larger — and we realised that what began as a modest design was turning into a full-scale language.

2.2 Using an IDL

Of course, we are not the first to encounter these difficulties. The standard solution is to use a separate Interface Definition Language (IDL) to describe the signatures of procedures that are to be called across the border. IDLs are rich and complicated, for precisely the reasons described above, but they are at least somewhat standardised and come with useful tools. We focus on the IDL used to describe COM interfaces [10], which is closely based on DCE IDL[7]. Another popular IDL dialect is the one defined by OMG as part of the CORBA specification[11], and we intend to provide support for this using the translation from OMG to DCE IDL defined by [12, 13].

Like COM, but unlike CORBA[1], we take the view that the IDL for a foreign procedure defines a *language-independent, binary interface to the foreign procedure* — a sort of *lingua franca*. The interface thus defined is supposed to be complete: it covers calling convention, data format, and allocation rules. It may be necessary to generate stub code on both sides of the border, to marshal parameters into the IDL-mandated format, and then on into the format demanded by the foreign procedure. But these two chunks of marshaling code can be generated separately, each by a tool specialised to its host language. By design, however, IDL's binary conventions are more or less identical to C's, so marshaling on the C side is hardly ever necessary.

Here, for example, is the IDL describing the interface to a function `foo`:

```
int foo( [out] long* l
       , [string, in] char* s
       , [in, out] double* d
       );
```

[1]CORBA does not define a binary interface. Rather, each ORB vendor provides a *language binding* for a number of supported languages. This language binding essentially provides the marshaling required to an ORB-specific common calling convention. If you want to use a language that the ORB vendor does not support, you are out of luck.

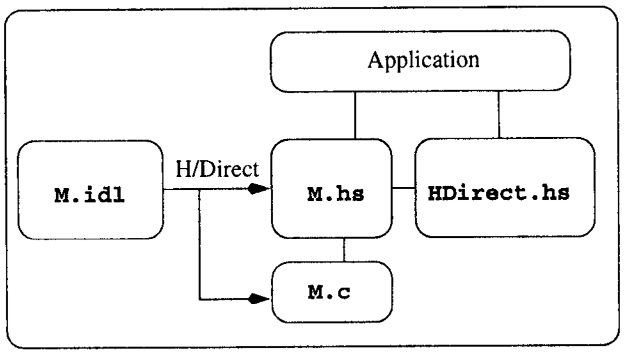

Figure 1: The big picture

The parts in square brackets are called *attributes*. In this case they describe the mode of each parameter, but there are a rich set of further attributes that give further (and often essential) information about the type of the parameters. For example, the string attribute tells that the parameter s points to a null-terminated array of characters, rather than pointing to a single character.

2.3 Overview

The "big picture" is given by Figure 1. The interface between Haskell and the foreign language is specified in IDL. This IDL specification is read by *H/Direct*, which then produces Haskell and C^2 source files files containing Haskell and C *stub code.*

H/Direct can generate stub code that allows Haskell to call C, or C to call Haskell. It can also generate stub code that allows Haskell to create and invoke COM components, and that allows COM components to be written in Haskell. Much of the work in all four cases concerns the marshaling of data between C and Haskell, and that is what we concentrate in this paper.

Since *H/Direct* generates Haskell source code, how does it express the actual foreign-language call (or entry for the inverse case)? We have extended Haskell with a foreign declaration that asks the Haskell implementation to generate code for a foreign-language call (or entry) [2]. The foreign declaration deals with the most primitive layer of marshaling, which is necessarily implementation dependent; *H/Direct* generates all the implementation-independent marshaling.

To make all this concrete, suppose we have the following IDL interface specification:

```
typedef struct { int x,y; } Point;

void Move( [in,out,ref] Point* p );
```

If asked to generate stub code to enable Haskell to call function Move, *H/Direct* will generate the following (Haskell) code:

^{2}For the sake of definiteness we concentrate on C as the foreign language in this paper.

```
data Point = Point { x,y::Int }
marshalPoint :: Point -> IO (Ptr Point)
marshalPoint = ...

unmarshalPoint :: Ptr Point -> IO Point
unmarshalPoint = ...

move :: Point -> IO Point
move p =
  do{ a <- marshalPoint p
    ; primMove a
    ; r <- unmarshalPoint a
    ; hdFree
    ; return r
    }
foreign import stdcall "Move"
  primMove :: Ptr Point -> IO ()
```

This code illustrates the following features:

- For each IDL declaration, *H/Direct* generates one or more Haskell declarations.
- From the IDL procedure declaration Move, *H/Direct* generates a Haskell function move whose signature is intended to be "what the user would expect". In particular, the Haskell type signature is expressed using *"high-level" types*; that is, Haskell equivalents of the IDL types. For example, the signature for move uses the Haskell record type Point. The translation for a procedure declaration is discussed in Section 3.
- The body of the procedure marshals the parameters into their *"low-level" types*, before calling the "low-level" Haskell function primMove. The latter is defined using a foreign declaration; the Haskell implementation generates code for the call to the C procedure Move. Section 4 specifies the high-level and low-level type corresponding to each IDL type.
- A "low-level" type is still a perfectly first-class Haskell type, but it has the property that it can trivially be marshaled across the border. There is fixed set of primitive "low-level" types, including Int, Float, Char and so on. Addr is a low-level type that holds a raw machine address. The type constructor Ptr is just a synonym for Addr:

  ```
  type Ptr a = Addr
  addPtr :: Ptr a -> Int -> Ptr b
  ```

  The type argument to Ptr is used simply to allow *H/Direct* to document its output somewhat, by giving the "high-level" type that was marshaled into that Addr. Section 5 describes how high-level types are marshaled to and from their low-level equivalents.
- From an IDL typedef declaration, *H/Direct* generates a corresponding Haskell type declaration together with some marshaling functions. In general, a marshaling function transforms a "high-level" Haskell value (in this case Point) into a "low-level" Haskell value (in this case Ptr Point). These marshaling functions are in the IO monad because, as we shall see, they often work imperatively by allocating some memory and explicitly filling it in, so as to construct a memory

layout that matches the interface specification. The translations for `typedef` declarations are discussed in Section 6.

- The function `hdFree :: IO ()` simply releases all the memory allocated by the marshaling functions.

So much for our example. The difficulty is that IDL is a complex language, so it is not always straightforward to guess the Haskell type that will correspond to a particular IDL type, nor to generate correct marshaling code. (The former is important to the programmer, the latter only to *H/Direct* itself.) Our goal in this paper is to give a systematic translation of IDL to Haskell stub code.

To simplify translation we assume that the IDL source is brought into a standard form, that is, we factor the translation into a translation of full IDL to a core subset and a translation from core IDL to Haskell. In particular, we assume that: out parameters always have an explicit "*", the pointer default is manifested in all pointer types, and all enumerations have value fields. (The details are unimportant.)

IDL is a large language, and space precludes giving a complete translation here. We do not even give a syntax for IDL, relying on the left-hand sides of the translation rules to specify the syntax we treat. However, the framework we give here is sufficient to treat the whole language, and our implementation does so.

3 Procedure declarations

The translation function $\mathcal{D}[\![\ ]\!]$ maps an IDL declaration into one or more Haskell declarations. We begin with IDL procedure declarations. To start with, we concentrate on allowing Haskell to call C; we discuss other variants in Section 7. Here is the translation rule for procedure declarations:

```
D⟦t_res f([in]t_in, [out]t_out, [in,out]t_inout)⟧
↦
T⟦f⟧ :: T⟦t_in⟧ -> T⟦t_inout⟧
     -> IO (T⟦t_out⟧,T⟦t_inout⟧,T⟦t_res⟧)
N⟦f⟧ = \m -> \n ->
  do { a <- M⟦t_in⟧ m
     ; b <- O⟦t_out⟧
     ; c <- M⟦t_inout⟧ n
     ; r <- primN⟦f⟧ a b c
     ; x  <- U⟦t_out⟧ b
     ; y  <- U⟦t_inout⟧ c
     ; z   <- U⟦t_res⟧ r
     ; hdFree
     ; return (x,y,z)
     }

  foreign import stdcall primN⟦f⟧
    :: B⟦t_in⟧ -> B⟦t_out⟧ -> B⟦t_inout⟧
    -> IO B⟦t_res⟧
```

Despite our claim of formality, the fully formal version of this rule has an inconvenient number of subscripts. Instead, we illustrate by giving one parameter of each mode ([in], [out], and [in, out]); more complex cases are handled exactly analogously. The translation produces a Haskell function that takes one argument for each IDL [in] or [in, out] parameter, and returns one result of each IDL [out] or [in, out] parameter, plus one result for the IDL result (if any). In general, foreign functions can perform side effects, so the result type is in the IO monad. We are considering adding a (non- standard) attribute [pure], that declares the procedure to have no side effects; in this case, the Haskell procedure can simply return a tuple rather than an IO type.

t :	b	*basic type*
|	n	*type names*
|	[{*attr*} +]*t**	*pointer type*
attr :	**unique** | **ref** | **ptr**	
	| **string** | **size_is(e)**	

Figure 2: IDL type syntax

The generic translation for procedure declaration uses several auxiliary translation schemes:

- The translation scheme $\mathcal{T}[\![t]\!]$ gives the "high-level" Haskell type corresponding to the IDL type t.
- The translation scheme $\mathcal{N}[\![n]\!]$ does the name mangling required to translate IDL identifiers to valid Haskell identifiers. For example, it accounts for the fact that Haskell function names must begin with a lower-case letter.
- The translation scheme $\mathcal{B}[\![t]\!]$ gives the "low-level" Haskell type corresponding to the IDL type t.
- The translation scheme $\mathcal{M}[\![t]\!]$ :: $\mathcal{T}[\![t]\!]$ -> IO $\mathcal{B}[\![t]\!]$ generates Haskell code that marshals a value of IDL type t from its high-level type $\mathcal{T}[\![t]\!]$ to its low-level form $\mathcal{B}[\![t]\!]$. This is used to marshal all the in-parameters of the procedure ([in] and [in,out]).
- The translation scheme $\mathcal{U}[\![t]\!]$ :: $\mathcal{B}[\![t]\!]$ -> IO $\mathcal{T}[\![t]\!]$ generates Haskell code that unmarshals a value of IDL type t. This is used to unmarshal all the out-parameters of the procedure, and its result (if any). $\mathcal{M}[\![\ ]\!]$ and $\mathcal{U}[\![\ ]\!]$ are mutual inverses (up to memory allocation).
- In addition, for [out] parameters the caller is required to allocate a location to hold the result. $\mathcal{O}[\![$[*attr*]*t**$]\!]$ :: IO (Ptr $\mathcal{B}[\![t]\!]$) is Haskell code that allocates enough space to contain a value of IDL type t.

4 Mapping for types

Next, we turn our attention to the translations $\mathcal{T}[\![\ ]\!]$ and $\mathcal{B}[\![\ ]\!]$ that translate IDL types to Haskell types. The syntax of IDL types that we treat is given in Figure 2, while Figure 3 gives their translation into Haskell types. We deal with user-defined structured types later, in Section 6.

Translating base types, which have direct Haskell analogues, is easy. The high-level and low-level type translations coincide, except that the high-level representation of IDL's 8-bit

$\mathcal{B}[\![\texttt{short}]\!]$	$\mapsto$	`Int32`
$\mathcal{B}[\![\texttt{unsigned short}]\!]$	$\mapsto$	`Word32`
$\mathcal{B}[\![\texttt{float}]\!]$	$\mapsto$	`Float`
$\mathcal{B}[\![\texttt{double}]\!]$	$\mapsto$	`Double`
$\mathcal{B}[\![\texttt{char}]\!]$	$\mapsto$	`Word8`
$\mathcal{B}[\![\texttt{wchar}]\!]$	$\mapsto$	`Char`
$\mathcal{B}[\![\texttt{boolean}]\!]$	$\mapsto$	`Bool`
$\mathcal{B}[\![\texttt{void}]\!]$	$\mapsto$	`()`
$\mathcal{B}[\![\texttt{[}attr\texttt{]}\ t\texttt{*}]\!]$	$\mapsto$	`Ptr` $\mathcal{T}[\![t]\!]$
$\mathcal{T}[\![\texttt{char}]\!]$	$\mapsto$	`Char`
$\mathcal{T}[\![b]\!]$	$\mapsto$	$\mathcal{B}[\![b]\!]$
$\mathcal{T}[\![n]\!]$	$\mapsto$	$\mathcal{N}[\![n]\!]$
$\mathcal{T}[\![\texttt{[ref]}\ t\texttt{*}]\!]$	$\mapsto$	$\mathcal{T}[\![t]\!]$
$\mathcal{T}[\![\texttt{[unique]}\ t\texttt{*}]\!]$	$\mapsto$	`Maybe` $\mathcal{T}[\![t]\!]$
$\mathcal{T}[\![\texttt{[ptr]}\ t\texttt{*}]\!]$	$\mapsto$	`Ptr` $\mathcal{T}[\![t]\!]$
$\mathcal{T}[\![\texttt{[string]char*}]\!]$	$\mapsto$	`String`
$\mathcal{T}[\![\texttt{[size\_is(}v\texttt{)]}\ t\texttt{*}]\!]$	$\mapsto$	`[`$\mathcal{T}[\![t]\!]$`]`

Figure 3: Type translations

characters is Haskell's 16 bit `Char` type. To give more precise mapping we have extended Haskell with new base types: `Word8`, `Word16`, and so on. Similarly, IDL type names are translated to the (Haskell-mangled) name of the corresponding Haskell type.

Matters start to get murkier when we meet pointers. Since a pointer is always passed to and from C as a machine address, the low-level translation of all pointer types is simply a raw machine address:

$$\mathcal{B}[\![\texttt{[}attr\texttt{]}\ t\texttt{*}]\!] \mapsto \texttt{Ptr}\ \mathcal{T}[\![t]\!]$$

(Recall that `Ptr` t is just an abbreviation for `Addr`, but the `Ptr` form is somewhat more informative.)

In contrast, the high-level translation of pointers depends on what type of pointer is concerned. IDL has no fewer than five kinds of pointer, distinguished by their attributes! We treat them one at a time (refer in each case to Figure 3):

- A value of IDL type `[ref]`t`*` is the unique pointer, or indirection, to a value of type t. A value of type `[ref]`t`*` should be marshalled by copying the structure over the border. Since pointers are implicit in Haskell, the corresponding high-level Haskell type is just $\mathcal{T}[\![t]\!]$.

- The IDL type `[unique]`t`*` is exactly the same as `[ref]`t`*`, except that the pointer can be `NULL`. The natural way to represent this possibility in Haskell is using the `Maybe` type. The latter is a standard Haskell type defined like this:

```
data Maybe a = Nothing | Just a
```

- An IDL value of type `[ptr]`t`*` is the address of a value that might be shared, and might contain cycles. It is far from clear how such a thing should be marshaled, so we adopt a simple convention:

$$\mathcal{T}[\![\texttt{[ptr]}\ t\texttt{*}]\!] \mapsto \texttt{Ptr}\ \mathcal{T}[\![\texttt{t}]\!]$$

  That is, `[ptr]` values are not moved across the border at all. Instead they are represented by a value of type `Ptr` $\mathcal{T}[\![\texttt{t}]\!]$, a raw machine address.

  This is often useful. For a start, some libraries implement an abstract data type, in which the client is expected to manipulate only pointers to the values. Similarly, COM interface pointers should be treated simply as addresses. Finally, some operating system procedures (notably those concerned with windows) return such huge structures that a client might want to marshal them back selectively.

- A value of type `[string]char*` is the address of a null-terminated sequence of characters. (Contrast `[ref]char*`, which is the address of a single character.) The corresponding Haskell type is, of course, `String`. The `[string]` attribute applies to the following array types `char`, `byte`, `unsigned short`, `unsigned long`, `structs` with `byte` (only!) fields and, in Microsoft-only IDL, `wchar`.

- Sometimes a procedure takes a parameter that is a pointer to an array of values, where another parameter of the procedure gives the size of the array. (CORBA IDL calls such arguments "sequences".) For example:

```
void DrawPolygon
  ( [in,size_is(nPoints)] Point* points
  , [in] int nPoints
  );
```

  The `[size_is(nPoints)]` attribute tells that the second parameter, `nPoints`, gives the size of the array. (This is quite like the `[string]` case, except that the size of the array is given separately, whereas strings have a sentinel at the end.) There is a second variant in which `nPoints` is a static constant, rather than the name of another parameter.

  At the moment we translate an IDL array to a Haskell list, but another possibility would be to translate it to a Haskell array. Different choices are probably "right" in different situations; perhaps we need a non-standard attribute to express the choice.

While each of these variants has a reasonable rationale, we have found the plethora of IDL pointer types to be a rich source of confusion. The translations in Figure 3 look innocuous enough, but we have found them extremely helpful in clarifying and formalising just exactly what the translation of an IDL type should be.

Even if the translations are not quite "right" (whatever that means), we now have a language in which to discuss variants. For example, it may eventually turn out that the IDL `[ptr]` attribute is conventionally used for subtly different purposes than the ones we suggest above. If so, the translations can readily be changed, and the changes explained to programmers in a precise way.

5 Marshaling

In the translation of the IDL type signature for a procedure (Section 3), we invoked marshaling functions $\mathcal{M}[\![\ ]\!]$ and $\mathcal{U}[\![\ ]\!]$ for each of the types involved. Now that we have defined the high and low-level translations of each type, the marshaling code is relatively easy to define. In this section we define these marshaling functions.

Marshaling a structured value consists, as we shall see, of two steps: allocate some memory in the *parameter-marshaling area* to hold the value, and then actually marshal the Haskell value into that memory. The translations are much more elegant if we define auxiliary schemes, $\mathcal{W}[\![\ ]\!]$ and $\mathcal{R}[\![\ ]\!]$, that perform this "by-reference" marshaling. We also need a number of functions to manipulate the parameter-marshaling area. More precisely:

$\mathcal{W}[\![t]\!]$ `:: Ptr` $\mathcal{T}[\![t]\!]$ `->` $\mathcal{T}[\![t]\!]$ `-> IO ()` marshals its second argument into the memory location(s) pointed to by its first argument; the latter is a raw machine address.

$\mathcal{R}[\![t]\!]$ `:: Ptr` $\mathcal{T}[\![t]\!]$ `-> IO` $\mathcal{T}[\![t]\!]$ unmarshals a value of IDL type t out of memory location(s) pointed to by its argument. $\mathcal{W}[\![\ ]\!]$ and $\mathcal{R}[\![\ ]\!]$ are mutually inverse (up to memory allocation).

$\mathcal{S}[\![t]\!]$ `:: Int` is the number of bytes occupied by an IDL value of type t. The function $\mathcal{O}[\![\ ]\!]$, mentioned in Section 3, is defined thus:

$$\mathcal{O}[\![\texttt{[}attr\texttt{]}\,t*]\!] \;\mapsto\; \texttt{hdAlloc}\ \mathcal{S}[\![t]\!]$$

`hdAlloc :: Int -> IO (Ptr a)` allocates the specified number of bytes in the parameter-marshaling area, returning a pointer to the allocated area.

`hdWrite`*b* `:: Ptr` $\mathcal{T}[\![\text{b}]\!]$ `->` $\mathcal{T}[\![\text{b}]\!]$ `-> IO ()`, where b is a basic type, marshals a value of IDL type b into the specified memory location(s).

`hdRead`*b* `:: Ptr` $\mathcal{T}[\![\text{b}]\!]$ `-> IO` $\mathcal{T}[\![\text{b}]\!]$, where b is a basic type, unmarshals a value of IDL type t.

`hdFree :: IO ()` frees the whole parameter-marshaling area.

With these definitions in mind, Figure 4 gives the marshaling schemes. We omit the schemes for `[size_is]` because it is tiresomely complicated. Apart from that, the translations are easy to read:

- For basic types there is no marshaling to do, except that we must convert between the 16-bit Haskell `Char` and 8-bit IDL `char` types.
- Marshaling a `typedef`'d type can be done by invoking its marshaling function.
- Marshaling a `[ref]` pointer is done by allocating some memory with `hdAlloc`, and then marshaling the value into it with $\mathcal{W}[\![\ ]\!]$. Unmarshaling is similar, except that there is no allocation step; we just invoke $\mathcal{R}[\![\ ]\!]$.
- Dealing with `[unique]` pointers is similar, except that we have to take account of the possibility of a `NULL` value.

Again, it is very helpful to have a precise language in which to discuss these translations. Though they look simple, we can attest that it is very easy to get confused by pointers to pointers to things, and we have far greater confidence in

```
M[[t]] :: T[[t]] -> IO B[[t]]

M[[char]]              |->  marshalChar
M[[b]]                 |->  return
M[[n]]                 |->  marshaln
M[[[ref] t*]]          |->  \x ->
                            do{ px <- hdAlloc S[[t]]
                              ; W[[t]] px x}
M[[[unique] t*]]       |->  \x ->
                            case x of
                              Nothing -> return nullPtr
                              Just y -> M[[[ref] t*]] y
M[[[ptr] t*]]          |->  return
M[[[string] t*]]       |->  marshalString
```

```
W[[t]] :: Ptr T[[t]] -> T[[t]] -> IO ()

W[[b]]                 |->  hdWriteb
W[[[attr] t*]]         |->  \p x ->
                            do{ a <- M[[[attr] t*]] x
                              ; hdWriteAddr p a}
```

```
U[[t]] :: B[[t]] -> IO T[[t]]

U[[char]]              |->  unmarshalChar
U[[b]]                 |->  return
U[[n]]                 |->  unmarshaln
U[[[ref] t*]]          |->  R[[t]]
U[[[unique] t*]]       |->  \p ->
                            if p == nullPtr then
                              return Nothing
                            else
                              do{ x <- R[[t]] p
                                ; return (Just x)}
U[[[ptr] t*]]          |->  return
U[[[string] t*]]       |->  unmarshalString
```

```
R[[t]] :: Ptr T[[t]] -> IO T[[t]]

R[[b]]                 |->  hdReadb
R[[[attr] t*]]         |->  \p ->
                            do{ a <- hdReadAddr p
                              ; U[[[attr] t*]] a}
```

Figure 4: The marshaling schemes

t :	`t[e]`	*array type*
|	`enum {` $tag_1 = v_1, \ldots, tag_n = v_n$`}`	*enumeration*
|	`struct` *tag* `{` $f_1 : t_1; \ldots; f_n : t_n;$ `}`	*record type*
|	`union` tag_1 `switch (` $b\ tag_2$ `) {` `case` v_1`:`t_1 f_1`;` ... `case` v_n`:`t_n f_n`;}`	*union type*

Figure 5: IDL constructed type syntax

our implementation as a result of writing the translations formally.

One might wonder about the run-time cost of all this data marshalling. Indeed, historically foreign-language interfaces have taken it for granted that data is *not* copied across the border. However, such non-marshalling interfaces are extremely restrictive: they require the two languages to share common data representations to the bit level, and to share a common address space. In moving decisively towards IDL-based component-based programming, the industry has accepted the performance costs of marshalling in exchange for its flexibility. This in turn discourages very fine-grain, intimate interaction between components with many border-crossings, instead encouraging a coarser-grain approach. We are happy to adopt this trend, because there is no way to make (lazy) Haskell and C share data representations.

6 Type declarations

On top of the primitive base types, IDL supports the definition of a number of constructed types. For example

```
typedef int trip[3];
typedef struct TagPoint { int x,y; } Point;
typedef enum { Red=0, Blue=1, Green=2 } RGB;
typedef union _floats switch (int ftype) {
  case 0: float  f;
  case 1: double d;
} Floats;
```

which declares array, record, enumeration and union (or sum) types, respectively. Figure 5 shows the syntax of IDL's constructed types.

The translation provides rules for converting between IDL constructed types into corresponding Haskell representations. To ease the task of defining this type mapping, we assume that each constructed type appears as part of an IDL type declaration. In general, a type declaration has the following form:

`typedef` *t name*`;`

declaring *name* to be a synonym for the type *t*, which is either a base type or one of the above constructed types. A type declaration for an IDL type *t* gives rise to the definition of the following Haskell declarations:

- A Haskell type declaration for the Haskell type $\mathcal{N}[\![name]\!]$, such that $\mathcal{T}[\![name]\!] = \mathcal{N}[\![name]\!]$.
- `marshal`$\mathcal{N}[\![name]\!]$ `::` $\mathcal{T}[\![name]\!]$ `-> IO` $\mathcal{B}[\![t]\!]$ which implements the $\mathcal{M}[\![\ ]\!]$ scheme for converting from the Haskell representation $\mathcal{T}[\![t]\!]$ to the IDL type *t*.
- `unmarshal`$\mathcal{N}[\![name]\!]$ `::` $\mathcal{B}[\![t]\!]$ `-> IO` $\mathcal{T}[\![name]\!]$ which implements the dual $\mathcal{U}[\![\ ]\!]$ scheme for unmarshaling.
- `marshal`$\mathcal{N}[\![name]\!]$`At :: Ptr` $\mathcal{B}[\![t]\!]$ `->` $\mathcal{T}[\![name]\!]$ `-> IO ()` for performing by-reference marshaling of the constructed type.
- `unmarshal`$\mathcal{N}[\![name]\!]$`At :: Ptr` $\mathcal{B}[\![t]\!]$ `-> IO` $\mathcal{T}[\![name]\!]$ which implements the $\mathcal{R}[\![\ ]\!]$ scheme for unmarshaling a constructed type by-reference.
- `sizeof`$\mathcal{N}[\![name]\!]$ `:: Int`, a constant holding the size of the external representation of the type (in 8-bit bytes.)

The general rules for converting type declarations into Haskell types is presented in Figure 6. Here is what they generate when applied:

- In the case of a type declaration for a base type, this merely defines a type synonym. For example

  ```
  typedef int year;
  ```

  is translated into the type synonym

  ```
  type Year = Int
  ```

  plus marshaling functions for `Year`.
- For a record type such as `Point`:

  ```
  typedef struct TagPoint {int x,y;} Point;
  ```

  generates a single constructor Haskell data type:

  ```
  data Point = TagPoint { x:: Int, y::Int }
  ```

  In addition to this, the $\mathcal{D}[\![\ ]\!]$ scheme generates a collection of marshaling functions, including `marshalPoint`:

  ```
  marshalPoint :: Point -> IO (Ptr Point)
  marshalPoint (Point x y) =
    do{ ptr <- hdAlloc sizeofPoint
      ; let ptr1 = addPtr ptr 0
      ; marshalintAt ptr1 x
      ; let ptr2 = addPtr ptr1 sizeofint
      ; marshalintAt ptr2 y
      ; return ptr
      }
  ```

  It marshals a `Point` by allocating enough memory to hold the external representation of the point. The size of the record type is computed as follows:

  ```
  sizeofPoint :: Int32
  sizeofPoint = structSize [sizeofint,sizeofint]
  ```

```
$\mathcal{D}[\![$typedef $t$ $name$;$]\!]$
 $\mapsto$ type $\mathcal{N}[\![name]\!]$ = $\mathcal{T}[\![t]\!]$
  marshal$\mathcal{N}[\![name]\!]$ = marshal$\mathcal{T}[\![t]\!]$
  marshal$\mathcal{N}[\![name]\!]$At = marshal$\mathcal{T}[\![t]\!]$At
  unmarshal$\mathcal{N}[\![name]\!]$ = unmarshal$\mathcal{T}[\![t]\!]$
  unmarshal$\mathcal{N}[\![name]\!]$At = unmarshal$\mathcal{T}[\![t]\!]$At
  sizeof$\mathcal{N}[\![name]\!]$ = $\mathcal{S}[\![t]\!]$

$\mathcal{D}[\![$typedef $t$ $name$[$dim$];$]\!]$
 $\mapsto$ type $\mathcal{N}[\![name]\!]$ = [ $\mathcal{T}[\![t]\!]$ ]
  marshal$\mathcal{N}[\![name]\!]$ = marshalArray $dim$ marshal$\mathcal{T}[\![t]\!]$At
  marshal$\mathcal{N}[\![name]\!]$At = marshalArrayAt $dim$ marshal$\mathcal{T}[\![t]\!]$At
  unmarshal$\mathcal{N}[\![name]\!]$ = unmarshalArray $dim$ unmarshal$\mathcal{T}[\![t]\!]$At
  unmarshal$\mathcal{N}[\![name]\!]$At = unmarshalArrayAt $dim$ unmarshal$\mathcal{T}[\![t]\!]$At
  sizeof$\mathcal{N}[\![name]\!]$ = $dim$ * $\mathcal{S}[\![t]\!]$

$\mathcal{D}[\![$typedef struct $tag\{\ldots;$ $t$ $field$; $\ldots\}$ $name$;$]\!]$
 $\mapsto$
  data $\mathcal{N}[\![name]\!]$ = $\mathcal{N}[\![tag]\!]${ ...,$\mathcal{N}[\![$field$_i]\!]$ :: $\mathcal{T}[\![t_i]\!]$, ...}
  marshal$\mathcal{N}[\![name]\!]$ rec = do
   ptr <- hdAlloc $\mathcal{S}[\![name]\!]$
   marshal$\mathcal{N}[\![name]\!]$At ptr rec
   return ptr

  marshal$\mathcal{N}[\![name]\!]$At ptr ($\mathcal{N}[\![tag]\!]${ ... ,$\mathcal{N}[\![$field$_i]\!]$, ...} = do
   let ptr$_1$ = addPtr ptr 0
   ...
   let ptr$_i$ = addPtr ptr$_{i-1}$ $\mathcal{S}[\![t_{i-1}]\!]$
   $\mathcal{W}[\![t_i]\!]$ ptr$_i$ field$_i$
   ...
   return ()

  unmarshal$\mathcal{N}[\![name]\!]$ = unmarshal$\mathcal{N}[\![name]\!]$At

  unmarshal$\mathcal{N}[\![name]\!]$At ptr = do
   let ptr$_1$ = addPtr ptr 0
   ...
   let ptr$_i$ = addPtr ptr$_{i-1}$ $\mathcal{S}[\![t_{i-1}]\!]$
   $\mathcal{N}[\![$field$_i]\!]$ <- $\mathcal{R}[\![t_i]\!]$ ptr$_i$
   ...
   return ($\mathcal{N}[\![tag]\!]$ ... $\mathcal{N}[\![$field$_i]\!]$ ... )
  sizeof$\mathcal{N}[\![name]\!]$ = structSize [...,$\mathcal{S}[\![$field$_i]\!]$,...]

$\mathcal{D}[\![$typedef enum $\{\ldots,alt$ = $value,\ldots\}$ $name$;$]\!]$
 $\mapsto$
  data $\mathcal{N}[\![name]\!]$ = ... | $\mathcal{N}[\![alt]\!]$ |...
  marshal$\mathcal{N}[\![name]\!]$ x =
   case x of {...; $\mathcal{N}[\![alt]\!]$ -> $\mathcal{N}[\![value]\!]$;...}
  unmarshal$\mathcal{N}[\![name]\!]$ x =
   case x of {...; $\mathcal{N}[\![value]\!]$ -> return $\mathcal{N}[\![alt]\!]$;...}
  unmarshal$\mathcal{N}[\![name]\!]$At ptr = do
   v <- hdReadInt ptr
   unmarshal$\mathcal{N}[\![name]\!]$ v
  sizeof$\mathcal{N}[\![name]\!]$ = sizeofint
```

Figure 6: Translating declarations

where `structSize` is a (platform specific) function that computes the size of a `struct` given the field sizes.[3]

`Point`'s two fields are marshaled into the external representation of `Point` by calling the by-reference marshaler for the basic type `Int`, supplying a pointer that has been appropriately offset.

- For the union type example given at the start of Section 6, the following Haskell type is generated:

  ```
  data Floats = F Float | D Double
  ```

  together with actions for marshaling between the algebraic type and a union (omitting the type signatures for the by-reference marshalers):

  ```
  marshalFloats   :: Floats -> IO (Ptr Floats)
  unmarshalFloats :: Ptr Floats -> IO Floats
  ```

  The external representation of a union is normally a `struct` containing the discriminant and enough room to accommodate the largest member of the union. In the case of `Floats`, the external representation must be large enough to contain an `int` and a `double`.

- Enumerations have a direct Haskell equivalent as algebraic data types with nullary constructors. For example, the `RGB` declaration:

  ```
  typedef enum {red=0,green=1,blue=2} RGB;
  ```

  is translated into the Haskell type

  ```
  data RGB = Red | Green | Blue
  ```

  with concrete representation $\mathcal{B}[\![\mathtt{RGB}]\!] = \mathtt{Int32}$

  The marshaling actions simply map between the nullary constructors and `Int32`:

  ```
  marshalRGB :: RGB -> IO Int32
  marshalRGB nm =
    return (case nm of {
                Red   -> 0
                Green -> 1
                Blue  -> 2 })

  unmarshalRGB :: Int32 -> IO RGB
  unmarshalRGB v =
   case v of
     0 -> return Red
     1 -> return Green
     2 -> return Blue
     _ -> fail (userError ...)
  ```

Haskell data structures can contain shared sub-components, or even cycles. However, such sharing is not observable by a Haskell program, so the marshalling code cannot take account of it. DAGs are therefore marshalled just as if they were trees, and a cyclic data structure (which is indistinguishable from an infinite data structure) would make the marshaler fail to terminate.

[3]Similarly, a function that returns the offsets at which to marshal each field into is also provided. Due to lack of space, `marshalPoint` makes the simplifying assumption that structures contain no internal padding.

It might be possible to "fix" these shortcomings, but we are not unduly bothered about them. Rather than marshal complex data structures (whether or not they contain sharing) across the border, a better approach is usually to leave them where they are and instead marshal a pointer to the data structure. When a component technology such as COM is being used (Section 8), the right thing to do is to marshal an interface pointer, through which the client can access the data structure.

In short, if loss of sharing is a worry then you are probably marshalling too much data; we look forward to learning from experience whether this viewpoint is "right".

7 The inverse mapping

Once marshaling and unmarshaling functions are defined for each data type, it is not hard to reverse the mapping and build code that allows C to call Haskell. The translation for a `typedef` remains unchanged, but the translation for an IDL procedure declaration is reversed. Since the procedure is being implemented in Haskell, its `[in]`-parameters are unmarshaled, the Haskell procedure is called, its results are marshaled, and returned to the caller. (We omit the details, but the translation rule can be expressed just as we did in Section 3.) For example, the `Move` IDL declaration of that Section would be compiled to the following Haskell code:

```
foreign export stdcall "Move"
  primMove :: Ptr Point -> IO ()

primMove a =
  do { p <- unmarshalPoint a
     ; q <- move p
     ; marshalPointAt a q
     ; return ()
     }

move :: Point -> IO Point
move = error "Not yet implemented"
```

The `foreign export` declaration asks the Haskell compiler to make `Move` externally callable with a `stdcall` interface. `primMove` does the marshaling, before calling `move`, which should be provided by the programmer.

8 Talking to COM

We are also interested in allowing Haskell programs to create and invoke COM objects, and in allowing a Haskell program to be sealed up inside a COM object. This too is a straightforward extension. There are a couple of wrinkles, however:

- COM methods conventionally return a value of type `HRESULT`, which is used to signal exceptional conditions. *H/Direct* "knows" about `HRESULT` and reflects its exceptional values into exceptions in Haskell's `IO` monad.
- COM methods are invoked indirectly, through a vector table. To support this the Haskell `foreign` declaration has to be extended to allow indirect calls. For example, the Haskell-to-COM side looks like this:

```
foreign import stdcall
        dynamic primFoo :: Addr -> ..
```

The keyword `dynamic` replaces the static name of the foreign function, and the address of the function is instead passed as the first argument to `primFoo`. The `foreign export` case is similar.

- Lastly, there are several design choices concerning what the programmer has to write to implement a COM object. Does she write a collection of functions that take the object state as their first argument? Or does she write a single function that returns a record of all the methods of the object?

9 Status and conclusions

H/Direct is now our fourth attempt at a foreign-language interface for Haskell. The first was `ccall`, a limited and low-level extension roughly equivalent to `foreign import` [3]. The second was Green Card, which gradually turned into a domain-specific language [5]. The third was a pre-cursor to *H/Direct*, Red Card, which was specifically aimed at interfacing Haskell to COM objects, [4, 6]. *H/Direct* embodies the lessons we have learned: strive for implementation-independence; avoid inventing new languages; the customer is always right.

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.

We are well advanced on an implementation of *H/Direct*. We can parse and type-check the whole of Microsoft IDL, and can generate stubs that allow Haskell to call C and COM. We have not yet implemented the reverse mapping, but we expect to do so in the next few months.

Acknowledgements

We thank Conal Elliott for playing the vital role of Friendly Customer; much of our motivation derives from his desired applications. We thank EPSRC and Microsoft for their support, both of equipment and manpower. Erik Meijer would like to thank the PacSoft group at the Oregon Graduate Institute for their hospitality during the final phases of writing this paper.

References

[1] D. Beazley. SWIG and automated C/C++ scripting extensions. *Dr. Dobb's Journal*, February 1998.

[2] Sigbjorn Finne et al. A primitive foreign function interface for Haskell. In preparation, preliminary specification available from `http://www.dcs.gla.ac.uk/fp/software/hdirect`, March 1998.

[3] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *The 20th ACM Symposium on Principles of Programming Languages(POPL)*, pages 71–84, 1993.

[4] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of the 5th International Conference on Software Reuse*, 1998.

[5] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green Card: a foreign-language interface for Haskell. In *Proceedings of the Haskell Workshop*, 1997.

[6] Daan Leijen. Red Card: Interfacing Haskell with COM. Master's thesis, University of Amsterdam, 1998. `http://www.haskell.org/active/activehaskell.html`.

[7] X/Open Company Ltd. *X/Open Preliminary Specification X/Open DCE: Remote Procedure Call*, 1993.

[8] Microsoft. *SDK for Java*, 1998. `http://www.microsoft.com/java/`.

[9] Microsoft Press. *Developing for Microsoft Agent*, 1998.

[10] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[11] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.

[12] A Vogel and B Gray. Translating DCE IDL in OMG IDL and vice versa. Technical Report 22, CRC for Distributed Systems Technology, Brisbane, 1995.

[13] A Vogel, B Gray, and K Duddy. Understanding any IDL, lesson one: DCE and CORBA. In *Proceedings of SDNE'96*, 1996.