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.

This paper has been submitted to the International Conference on Functional Programming 1998 (ICFP'98).

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 [9], 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 un-marshals 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 marshalling, 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:

```java
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 `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 ever 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 [13, 12].

Like COM, but unlike CORBA¹, 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 clunks 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:

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

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.

¹CORBA 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.
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 \( H/\text{Direct} \), which then produces Haskell and C\(^2\) source files files containing Haskell and C stub code.

\( H/\text{Direct} \) 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 \( H/\text{Direct} \) 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; \( H/\text{Direct} \) generates all the implementation-independent marshaling.

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

```c
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 \( \text{Move} \), \( H/\text{Direct} \) will generate the following (Haskell) code:

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

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

move :: Point -> IO Point
move p =
```

```c
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, \( H/\text{Direct} \) generates one or more Haskell declarations.
- From the IDL procedure declaration \( \text{Move} \), \( H/\text{Direct} \) generates a Haskell function \( \text{move} \) 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 \( \text{move} \) uses the Haskell record type \text{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 \( \text{primMove} \). The latter is defined using a foreign declaration; the Haskell implementation generates code for the call to the C procedure \text{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 marshalled across the border. There is fixed set of primitive “low-level” types, including \text{Int}, \text{Float}, \text{Char}

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, or to generate correct marshalling code. (The former is important to the programmer; the latter only to \(H/\text{Direct}\) 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 enumeration 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:

\[
\mathcal{D}[\ ]:\ (\{\text{has|lin}, \ [\text{out}]\text{-}\text{out}, \ [\text{in}, \text{out}]\text{-}\text{inout}\}) \rightarrow T[\ ]\rightarrow T[\text{-}\text{inout}], T[\text{-}\text{res}]\rightarrow IO(T[\text{-}\text{in}], T[\text{-}\text{inout}], T[\text{-}\text{res}])
\]

\[
\mathcal{N}[\ ]:\ = \text{\textnormal{\textbackslash m} -> \text{\textnormal{\textbackslash n}}} \rightarrow \text{\textbackslash m}
\]

\[
\text{do}\ \{\text{a \leftarrow \mathcal{M}[\text{-}\text{in}]}\text{\ m} ; \text{b \leftarrow \mathcal{O}[\text{-}\text{out}]}; \text{c \leftarrow \mathcal{N}[\text{-}\text{inout}] n} ; \text{r \leftarrow \mathcal{F}[\text{-}\text{var}] a b c} ; \text{x \leftarrow \mathcal{U}[\text{-}\text{out}] b} ; \text{y \leftarrow \mathcal{U}[\text{-}\text{inout}] c} ; \text{z \leftarrow \mathcal{U}[\text{-}\text{res}] x} ; \text{\text{\textnormal{\textbackslash hdFree}}} ; \text{\text{return } (x,y,z)}\}
\]

\[
\text{foreign import stdcall \mathcal{F}[\text{-}\text{var}]}:: \mathcal{B}[\text{-}\text{in}] \rightarrow \mathcal{B}[\text{-}\text{out}] \rightarrow \mathcal{B}[\text{-}\text{inout}]\rightarrow IO \mathcal{B}[\text{-}\text{res}]
\]

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 \((\text{fin}\), \(\text{out}\), and \(\text{in}, \text{out}\)); more complex cases are handled exactly analogously. The translation produces a Haskell function that takes one argument for each IDL \([\text{in}]\) or \([\text{in}, \text{out}]\) parameter, and returns one result of each IDL \([\text{out}]\) or \([\text{in}, \text{out}]\) 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 \((\text{non-standard})\) attribute \text{\textnormal{\textbackslash export}}, 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.

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

\[
\begin{array}{c|c}
 t & b \quad \text{basic type} \\
 n & \text{type names} \\
 [(\text{attr}) +] \ast & \text{pointer type} \\
 \text{attr} & \text{unique | ref | ptr} \\
 \text{string} & \text{size_s(e)}
\end{array}
\]

![Figure 2: IDL type syntax](image)

- The translation scheme \(T[\ ]\) gives the “high-level” Haskell type corresponding to the IDL type \(t\).
- The translation scheme \(N[\ ]\) does the name mangling required to translate IDL identifiers to valid Haskell identifiers. For instance to account for the fact that Haskell function names must begin with a lower-case letter.
- The translation scheme \(B[\ ]\) gives the “low-level” Haskell type corresponding to the IDL type \(t\).
- The translation scheme \(M[\ ]\) generates Haskell code that marshals a value of IDL type \(t\) from its high-level type \(T[\ ]\) to its low-level form \(B[\ ]\). This is used to marshal all the in-parameters of the procedure \((\text{in} \text{and}\text{inout})\) are mutual inverses \((\text{upto memory allocation})\).
- In addition, for \([\text{out}]\) parameters the caller is required to allocate a location to hold the result. \(C[\ ]\) is Haskell code that allocates enough space to contain a value of IDL type \(t\).

We will define these functions in detail in Section 4, but first we deal with type declarations.

4 Mapping for types

Next, we turn our attention to the translations \(T[\ ]\) and \(B[\ ]\) that translate IDL types to Haskell types, which are given in Figure 3.

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

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

\[
B[\ast] \rightarrow \text{\textnormal{\textbackslash hdFree}} T[\ ]
\]
is/(
[54x421]treat them one at a time /(refer in each case to Figure /3/: /
form is somewhat more informative./) /(Recall that
The IDL type
A value of IDL type
B[short] → Int32
B[unsigned short] → Word32
B[float] → Float
B[double] → Double
B[char] → Word8
B[short] → Char
B[bool] → Bool
B[size] → ()
B[nptr] → * T

T[char] → Char
T[i] → N
T[ref]* → T[i]
T[unique]* → Maybe T[i]
T[ptr]* → Ptr T[i]
T[string] → String
T[size_in] → T[i]

Figure 3: Type translations

(Recall that * 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]* is the unique pointer,
or indirection, to a value of type t. Since pointers
are implicit in Haskell, the corresponding high-level
Haskell type is just T'[t].

• The IDL type [unique]* is exactly the same as
[ref]*, 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]* 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 marshalled,
so we adopt a simple convention:

T'[ptr]* → Ptr T'[t]

That is, *ptr values are not moved across the border
at all. Instead they are represented by a value of type
Ptr T'[t], a raw machine address.

This is often useful. For a start, some libraries im-
plement 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] is the address of a
null-terminated sequence of characters. (Contrast
[string], 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,
structures 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. For example:

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

The [size_is(nPoints)] attribute tells that the sec-
ond 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.) We translate arrays to
Haskell lists.

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 in-
nocuous enough, but we have found them extremely helpful
in clarifying and formalising just exactly what the transla-
tion of an IDL type should be.

Even if the translation are not quite "right" (whatever that
means), we now have a language in which to discuss vari-
ants. 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 transla-
tions can readily be changed, and the changes explained to
programmers in a precise way.

5 Marshalling

In the translation of the IDL type signature for a proce-
dure (Section 3), we invoked marshalling functions M[ ]
and U[ ], for each of the types involved. Now that we have
defined the high and low-level translations of each type, the
marshalling code is relatively easy to define. In this sec-
ction we define these marshalling functions. Lack of space
precludes us from giving complete details so we will con-
centrate mostly on marshalling basic types.

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

W[t] :: Ptr T'[t] → 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.
\[ R[t] \mapsto \text{Ptr } T[t] \rightarrow I0 \ T[t] \] unmarshals a value of IDL type \( t \) out of memory location(s) pointed to by its argument. \( W[I] \) and \( R[I] \) are mutually inverse (up to memory allocation).

\( S[t] \mapsto \text{Int is the number of bytes occupied by an IDL value of type } t. \) The function \( O[I] \), mentioned in Section 3, is defined thus:

\[ O[\text{[attr]}][\ast] \mapsto \text{hdAlloc } S[t] \]

\text{hdAlloc} : \text{Int } \rightarrow \text{I0 (Ptr a)} \text{ allocates the specified number of bytes in the parameter-marshalling area, returning a pointer to the allocated area.}

\text{hdWrite} : \text{Ptr } T[b] \rightarrow T[b] \rightarrow I0 () \text{, where } b \text{ is a basic type, marshals a value of IDL type } b \text{ into the specified memory location(s).}

\text{hdRead} : \text{Ptr } T[b] \rightarrow I0 T[b], \text{ where } b \text{ is a basic type, unmarshals a value of IDL type } t.

\text{hdFree} : I0 () \text{ frees the whole parameter-marshalling area.}

With these definitions in mind, Figure 4 gives the marshalling schemes. We omit the schemes for \([\text{size}][\text{attr}]\) because it is tiresomely complicated. Apart from that, the translations are easy to read:

- For basic types there is no marshalling to do, except that we must convert between the 16-bit Haskell Char and 8-bit IDL char types.

- Marshalling a typedef'd type can be done by invoking its marshalling function.

- Marshalling a [ref] pointer is done by allocating some memory with \text{hdAlloc}, and then marshalling the value into it with \( W[I] \). Unmarshalling is similar, except that there is no allocation step; we just invoke \( R[I] \).

- 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 points to things and we have far greater confidence in our implementation as a result of writing the translations formally.

### 6 Type declarations

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

```c
typedef int trip[3];
typedef struct TagPoint { int x,y; } Point;
typedef enum { Red=0, Blue=1, Green=2 } RGB;
typedef union floatS { float f;
    case 0: double d;
} FloatS;
```

![Figure 4: The marshalling schemes](image-url)
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:
  
  ```haskell
typedef int year;
```

is translated into the type synonym

```
type Year = Int
```

plus marshalling functions for Year.

- For a record type such as Point:

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

  generates a single constructor Haskell data type:

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

In addition to this, the D[ ] scheme generates a collection of marshalling functions, including `marshallPoint`:

```haskell
marshallPoint :: Point -> IO (Ptr Point)
marshallPoint (Point x y) =
do{ ptr <- hdAlloc sizeofPoint
  ; let ptr1 = addPtr ptr 0
  ; marshallInt ptr1 x
  ; let ptr2 = addPtr ptr1 sizeofInt
  ; marshallInt 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:

```haskell
sizeofPoint :: Int32
sizeofPoint = structSize [sizeofInt,sizeofInt]
```

where `structSize` is a [platform specific] function that computes the size of a struct given the field sizes. Point's two fields are marshalled into the external representation of Point by calling the by-reference marshaller 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:

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

```
```

plus actions for marshalling between the algebraic type and a union (omitting the type signatures for the by-reference marshalers):

```haskell
marshallFloats :: Floats -> IO (Ptr Floats)
unmarshallFloats :: 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 and `int` and a `double`.

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

```haskell
typedef enum 
    t color 
    { red, green, blue }
```

```haskell

type RGB = color
```

is translated into the type synonym

```
type RGB = Int
```

plus marshalling functions for RGB.
```c
typedef enum {name} tag;  
marshallN[name] = marshallT[t]  
marshalN[name]At = marshallT[t]At  
unmarshallN[name] = unmarshallT[t]  
unmarshallN[name]At = unmarshallT[t]At  
sizeofN[name] = S[t]

type name[dim];  
marshallN[name] = marshallArray dim marshallT[t]At  
marshalN[name]At = marshallArrayAt dim marshallT[t]At  
unmarshallN[name] = unmarshallArray dim unmarshallT[t]At  
unmarshallN[name]At = unmarshallArrayAt dim unmarshallT[t]At  
sizeofN[name] = dim * S[t]

type struct {tag(...; field; ...) name;}

data N[name] = N[tag] { ... , N[field], ... } : T[t], ...  
marshalN[name] rec = do  
ptr <- hdAlloc S[name]  
marshalN[name]At ptr rec  
return ptr  
marshallN[name]At ptr (N[tags] ... N[field], ... ) = do  
let ptr_i = addPtr ptr 0  
...  
let ptr_i = addPtr ptr_i S[t_i]  
W[i] ptr_i field_i  
...  
return ()  
unmarshallN[name] = unmarshallN[name]At  
unmarshallN[name]At ptr = do  
let ptr_i = addPtr ptr 0  
...  
let ptr_i = addPtr ptr_i S[t_i]  
N[field] <- R[i] ptr_i  
...  
return (N[tags] ... N[field] ... )  
sizeofN[name] = structSize [ ... , S[field], ... ]

type enum { ... , alt = value, ... } name;

data N[name] = ... | N[alt] | ...  
marshallN[name] x =  
case x of { ... , N[alt] -> N[value]; ... }  
unmarshallN[name] x =  
case x of { ... , N[value] -> return N[alt]; ... }  
unmarshallN[name]At ptr = do  
v <- hdReadInt ptr  
unmarshallN[name] v  
sizeofN[name] = sizeofint
```

Figure 6: Translating declarations
typedef enum {red=0, green=1, blue=2} RGB;

is translated into the Haskell type

data RGB = Red | Green | Blue

with concrete representation RGB = Int32

The marshalling actions simply map between the
nullary constructors and Int32:

marshallRGB :: RGB -> IO Int32
marshallRGB mm =
   return (case { Red -> 0; Green -> 1; Blue -> 2})

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

7 The inverse mapping

Once marshalling and un-marshalling 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
type remains unchanged, but the translation for an
IDL procedure declaration is reversed. Since the procedure
is being implemented in Haskell, its [in]-parameters are
un-marshalled, the Haskell procedure is called, its results
are marshalled, 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 import stdcall "Move"
primMove :: Addr Point -> IO ()
primMove a =
do { p <- unmarshallPoint a
   ; q <- move p
   ; marshallPointAt q p
   ; return ()
}

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

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

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

- 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?

8 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
da domain-specific language [5]. The third was a pre-cursor
to H/Interface, Red Card, which was specifically aimed at in-
terfacing Haskell to COM objects, [4, 6]. H/Interface
embodies the lessons we have learned: strive for implementa-
tion-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 de-
scription 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/Interface. 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/Interface.
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 sup-
port, 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


