Reading for Monday

Subsequent pages of this document contain the appropriate excerpts from the 3 papers, in the order of the bullet points listed below:

- H/Direct:
- **Read** sections 1 and 2: this sets up the problem and design space.
  - In section 2.3: **focus on understanding** IDL types versus Haskell types.
- **Read** the bullets on page 157: these explain the five IDL pointer types.
- **Attempt** to understand what `marshalPoint` on page 159 is doing.
- **Take note of** the main claim on the page after page 159.
- Stretching the storage manager:
- **Read** section 4 to understand *stable names* and *stable pointers*
  - **Read** section 4.4 *closely* to understand some garbage collection implications.
- **Read** section 5.6 to understand the memory management issue (not the solution).
- Calling hell from heaven and heaven from hell:
- **Read** the second bulleted point in the intro.
- **Read** section 3
  - **Read** section 3.3 *closely* (*stable pointers*)
  - **Read** section 3.5 *closely* (higher-order *callbacks*)
- The Lua Registry:
- **Read** section 27.3.1: "Lua offers a separate table, called the registry, that C code can freely use, but Lua code cannot access."

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.

ICFP '98 Baltimore, MD USA

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.

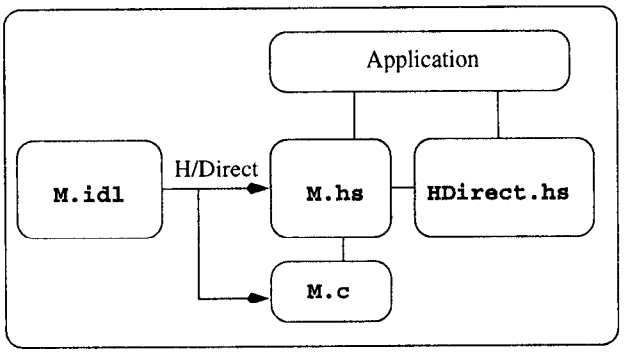

Figure 1: The big picture

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

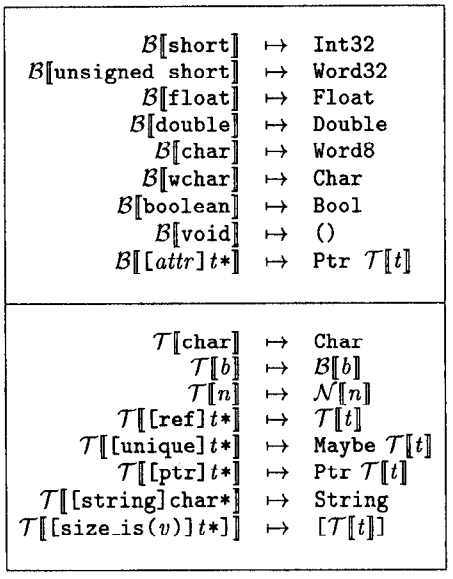

Figure 3: Type translations

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` $\mathcal{T}$[[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

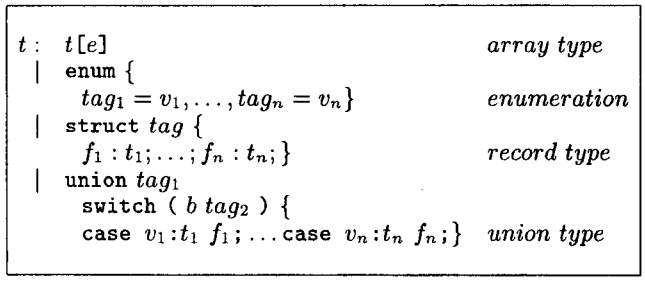

Figure 5: IDL constructed type syntax

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]
  ```

Section 9: main claim of H/Direct paper

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 misimplemented. 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.

Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell

Simon Peyton Jones[1], Simon Marlow[2], and Conal Elliott[3]

[1] Microsoft Research, Cambridge, simonpj@microsoft.com
[2] Microsoft Research, Cambridge, simonmar@microsoft.com
[3] Microsoft Research, Redmond, conal@microsoft.com

Abstract. Every now and then, a user of the Glasgow Haskell Compiler asks for a feature that requires specialised support from the storage manager. Memo functions, pointer equality, external pointers, finalizers, and weak pointers, are all examples.
We take memo functions as our exemplar because they turn out to be the trickiest to support. We present no fewer than four distinct mechanisms that are needed to support memo tables, and that (in various combinations) satisfy a variety of other needs.
The resulting set of primitives is undoubtedly powerful and useful. Whether they are *too* powerful is not yet clear. While the focus of our discussion is on Haskell, there is nothing Haskell-specific about most of the primitives, which could readily be used in other settings.

1 Introduction

"Given an arbitrary function `f`*, construct a memoised version of* `f`*; that is, construct a new function with the property that it returns exactly the same results as* `f`*, but if it is applied a second time to a particular argument it returns the result it computed the first time, rather than recomputing it."*

Surely this task should be simple in a functional language! After all, there are no side effects to muddy the waters. However, it is well known that this simple problem raises a whole raft of tricky questions. A memo table inherently involves a sort of "benign side effect", since the memo table is changed as a result of an application of the function; how should we accommodate this side effect in a purely-functional language? What does it mean for an argument to be "the same" as a previously encountered one? Does a memo function have to be strict? Efficient memo tables require at least ordering, and preferably hashing; how should this be implemented for arbitrary argument types? Does the memo function retain all past (argument,result) pairs, or can it be purged? Can the entire memo table ever be recovered by the garbage collector? And so on.

One "solution" is to build in memo functions as a primitive of the language implementation, with special magic in the garbage collector and elsewhere to deal with these questions. But this is unsatisfactory, because a "one size fits all" solution is unlikely to satisfy all customers. It would be better to provide a simpler set of primitives that together allowed a programmer to write a variety

P. Koopman and C. Clack (Eds.): IFL'99, LNCS 1868, pp. 37–58, 2000.

instead use some kind of lookup tree, based on ordering (not just equality) of the argument. That would in turn require that the argument type was ordered, thus changing `memo`'s type again:

```
memoOrd :: Ord a => (a -> b) -> a -> b
```

`memoOrd` can be implemented exactly as above, except that the lookup and insert functions become more complicated. We can do hashing in a very similar way. Notation apart, all of this is exactly how a Lisp programmer might implement memo functions. All we have done is to make explicit exactly where the programmer is undertaking proof obligations — a modest but important step.

4 Stable Names

Using equality, as we have done in `memoEq`, works OK for base types, such as `Int` and `Float`, but it becomes too expensive when the function's argument is (say) a list. In this case, we almost certainly want something like pointer equality; in exchange for the fast test we accept that two lists might be equal without being pointer-equal.

However, having only (pointer) equality would force us back to association lists. To do better we need ordering or a hash function. The well-known difficulty is that unless the garbage collector never moves objects (an excessively constraining choice), an object's address may change, and so it makes a poor hash key. Even the relative ordering of objects may change.

What we need is a cheap address-like value, or *name* that can be derived from an arbitrary value. This name should be *stable*, in the sense that it does not change over the lifetime of the object it names. With this in mind, we provide an abstract data type `StableName`, with the following operations:

```
data StableName a     -- Abstract

mkStableName   :: a -> IO (StableName a)
hashStableName :: StableName a -> Int

instance Eq  (StableName a)
instance Ord (StableName a)
```

The function `mkStableName` makes a stable name from any value. Stable names support equality (class `Eq`) and ordering (class `Ord`). In addition, the function `hashStableName` converts a stable name to a hash key.

Notice that `mkStableName` is in the `IO` monad. Why? Because two stable names might compare less-than in one run of the program, and greater-than in another run. Putting `mkStableName` in the `IO` monad is a standard trick that allows `mkStableName` to consult (in principle) some external oracle before deciding what stable name to return. In practice, we often wrap calls to `mkStableName` in an `unsafePerformIO`, thereby undertaking a proof obligation that the meaning

```
data SNMap k v -- abstract

newSNMap    :: IO (SNMap k v)
lookupSNMap :: SNMap k v -> StableName k -> IO (Maybe v)
insertSNMap :: SNMap k v -> StableName k -> v -> IO ()
removeSNMap :: SNMap k v -> StableName k -> IO ()
snMapElems  :: SNMap k v -> IO [(k,v)]
```

Fig. 1. Stable Name Map Library

of the program does not depend on the particular stable name that the system chooses.

Stable names have the following property: if two values have the same stable name, the two values are equal

$$(\dagger) \qquad \texttt{mkStableName}\ x = \texttt{mkStableName}\ y \quad \Rightarrow \quad x = y$$

This property means that stable names are unlike hash keys, where two keys might accidentally collide. If two stable names are equal, no further test for equality is necessary. An immediate consequence of (†) is this: if two values are not equal, their stable names will differ.

$$x \neq y \quad \Rightarrow \quad \texttt{mkStableName}\ x \ \neq \ \texttt{mkStableName}\ y$$

`mkStableName` is not strict; it does not evaluate its argument. This means that two equal values might not have the same stable name, because they are still distinct unevaluated thunks. For example, consider the definitions

```
p  = (x,x);  f1 = fst p;  f2 = snd p
```

So long as `f1` and `f2` remain unevaluated, `mkStableName f1` will return a different stable name than `mkStableName f2`[3].

It is easy to make `mkStableName` strict, by using Haskell's strict-application function "`$!`". For example, `mkStableName $! f1` and `mkStableName $! f2` would return the same stable name. Using strict application loses laziness, but increases sharing of stable names, a choice that only the programmer can make.

4.1 Using Stable Names for Memo Tables

Throughout the rest of this paper, we will make use of Stable Name Maps, an abstract data type that maps Stable Names to values (Figure 1). The implementation may be any kind of mutable finite map, or a real hash table (using `hashStableName`).

[3] A compiler optimisation might well have evaluated `f1` and `f2` at compile time, in which case the two calls would return the same stable name; another example of why `mkStableName` is in the `IO` monad.

Using stable names it is easy to modify our memo-table implementation to use pointer equality (strict or lazy) instead of value equality. We give only the code for the `apply` part of the implementation

```
applyStable :: (a -> b) -> SNMap a b -> a -> b
applyStable f tbl arg
  = unsafePerformIO ( do
      { sn  <- mkStableName arg
      ; lkp <- lookupSNMap tbl sn
      ; case lkp of
            Just result -> return result
            Nothing     -> do { let res = f arg
                              ; insertSNMap tbl sn res
                              ; return res
      }                       })
```

4.2 Implementing Stable Names

Our implementation is depicted in Figure 2. We maintain two tables. The first is a hash table that maps the address of an object to an offset into the second table, the *Stable Name Table*. If the address of a target changes during garbage collection, the hash table must be updated to reflect its new address. There are two possible approaches:

- Always throw away the old hash table and rebuild a new one after each garbage collection. This would slow down garbage collection considerably when there are a large number of stable names.
- In a generational collector, we have the option of partially updating the hash table during a minor collection. Only the entries for targets which have moved during the current GC need to be updated. This is the method used by our implementation.

Each slot in the Stable Name Table (SNT) corresponds to a distinct stable name. The stable name can be described by its offset in the SNT, and it is this offset that is used for equality and comparison of stable names.

However, we cannot simply use this offset as the value returned by `mkStableName`! Why not? Because in order to maintain (†) we must ensure that we never re-use a stable name to which the program still has access, *even if the object from which the stable name was derived has long since died*.

Accordingly, we represent a value of type `StableName a` by a *stable name object*, a heap-allocated cell containing the SNT offset. It is this object that is returned as the result of `mkStableName`. The entry in the SNT points to the corresponding stable name object, and also the object for which the stable name was created (the *target*).

Now entries in the SNT can be garbage-collected as follows. The SNT is not treated as part of the root set. Instead, when garbage collection is complete, we

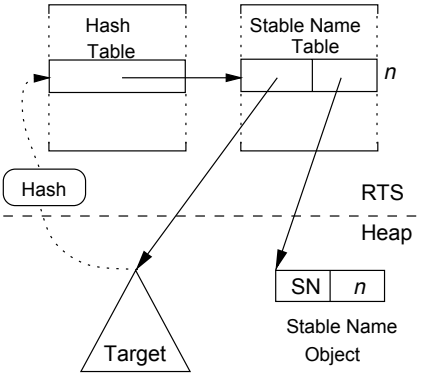

Fig. 2. Stable Name Implementation

scan the entries of the SNT that are currently in use. If an entry's stable name object is dead (not reachable), then it is safe to re-use the stable name entry, because the program cannot possibly "re-invent" it. For each stable name entry that is still live, we also need to update the pointers to the stable name object and the target, because a copying collector might have moved them.

Available entries in the SNT are chained on a free list through the stable-object-pointer field.

4.3 `hashStableName`

The `hashStableName` function satisfies the following property, for stable names a and b:

$$a = b \quad \Rightarrow \quad \texttt{hashStableName}\ a = \texttt{hashStableName}\ b$$

The converse is not true, however. Why? The call `hashStableName a` is implemented by simply returning the offset of the stable name `a` in the SNT. Because the `Int` value returned can't be tracked by the garbage collector in the same way as the stable name object, it is possible that calls to `hashStableName` on different stable names could return the same value. For example:

```
do { sn_a <- mkStableName a
   ; let hash_a = hashStableName sn_a
   ; sn_b <- mkStableName b
   ; let hash_b = hashStableName sn_b
   ; return (hash_a == hash_b)
   }
```

Assuming `a` and `b` are distinct objects, this piece of code could return `True` if the garbage collector runs just after the first call to `hashStableName`, because

the slot in the SNT allocated to `sn_a` could be re-used by `sn_b` since `sn_a` is garbage at this point.

4.4 Other Applications

An advantage of the implementation we have described is that we can use the very same pair of tables for two other purposes. When calling external libraries written in some other language, it is often necessary to pass a Haskell object. Since Haskell objects move around from time to time, we actually pass a *Stable Pointer* to the object. A stable pointer is a variant of a stable name, with slightly different properties:

1. It is possible to dereference a stable pointer to get to the target. This means that the existence of a stable pointer must guarantee the existence of the target.
2. Stable pointers are reference counted, and must be explicitly freed by the programmer. This is because a stable pointer can be passed to a foreign function, leaving no way for the Haskell garbage collector to track it.

We implement stable pointers using the same stable name technology. The stable name table already contains a pointer to the target of the stable name, hence (1) is easy. To support (2) we add a reference count to the SNT entry, and operations to increment and decrement it. The pointer to the target is treated as a root by the garbage collector if and only if the reference count is greater than zero.

We use exactly the same technology again for our parallel implementation of Haskell, Glasgow Parallel Haskell (GPH). GPH distributes a single logical Haskell heap over a number of disjoint address spaces [11]. Pointers between these sub-heaps go via stable names, thus allowing each sub-heap to be garbage collected independently. Weighted reference counting is used for global garbage collection [8].

The point here is simply that a single, primitive mechanism supports all three facilities: stable names, passing pointers to foreign libraries, and distributed heaps.

5 Weak Pointers

If a memoised function is discarded, then its memo table will automatically be garbage collected. But suppose that a memoised function is long-lived, and is applied to many arguments, many of which are soon discarded. This situation gives rise to a well-known space leak:

– Since the memo table contains references to all the arguments to which the function has ever been applied, those arguments will be reachable (in the eyes of the garbage collector) even though the function will never be applied to that argument again.

Section 5.6: the memory management issue

Another situation where we found weak pointers to be "just the right thing" is when referencing objects outside the Haskell heap via proxy objects (a proxy object is an object in the local heap that just contains a pointer to the foreign object).

Consider a structured foreign object, to which we have a proxy object in the Haskell heap. The garbage collector will track the proxy object in order that the foreign object can be freed when it is no longer referenced from Haskell (probably using a finalizer, see the next section). If we are given a pointer to a subcomponent of the foreign object, then we need a suitable way to keep the proxy for the *root* of the foreign object alive until we drop the reference to the subcomponent.

A weak pointer solves this problem nicely: the *key* points to a proxy for the subcomponent, and the *value* points to the proxy for the root. The entire foreign object will thereby be retained until all references to the subcomponent are dropped.

proxy for the *root* of the foreign object alive until we drop the reference to the subcomponent.

A weak pointer solves this problem nicely: the *key* points to a proxy for the subcomponent, and the *value* points to the proxy for the root. The entire foreign object will thereby be retained until all references to the subcomponent are dropped.

6 Finalization

We did not present code for purging the memo table of useless key/value pairs. Indeed, the whole idea is less than satisfactory, because it amounts to *polling* the keys to see if they have died. It would be better to receive some sort of *notification* when the key died.

Indeed, it is quite common to want to perform some sort of clean-up action when an object dies; such actions are commonly called *finalization*. If it were possible to attach a finalizer to the key, then when the key dies, the finalizer could delete the entry from the memo table. A particular key might be in many memo tables, so it is very desirable to be able to attach multiple finalizers to a particular object.

Finalizers are often used for *proxy objects* that encapsulate some external resource, such as a file handle, graphics context, malloc'd block, network connection, or whatever. When the object becomes garbage, the finalizer runs, and can close the file, release the graphics context, free the malloc'd block, etc. In some sense, these proxy objects are the dual to stable pointers (Section 4.4): they encapsulate a pointer from Haskell to some external world, while a stable pointer encapsulates a pointer from the external world into Haskell.

Finalizers raise numerous subtle issues. For example, does it matter which order finalizers run in, if several objects die "simultaneously" (whatever that means)? The finalizer may need to refer to the object it is finalizing, which presumably means "resurrecting" it from the dead. If the finalizer refers to the object, might that keep it alive, thereby vitiating the whole effect? If not, how does the finalizer get access to the object? How promptly do finalizers run? And so on. [3] gives a useful overview of these issues, and a survey of implementations.

6.1 A Design for Finalizers

In our experience, applications that use weak pointers almost always require some sort of finalization as well, so we have chosen to couple the two. We add the following two new functions:

```
mkWeak   :: k -> v -> Maybe (IO ()) -> IO (Weak v)
finalize :: Weak v -> IO ()
```

`mkWeak` is like `mkSimpleWeak`, except that it takes an extra argument, an optional finalization action. The call `(mkWeak k v (Just a))` has the following semantics:

Calling hell from heaven and heaven from hell

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
Microsoft Research Cambridge
simonpj@microsoft.com

Abstract

The increasing popularity of component-based programming tools offer a big opportunity to designers of advanced programming languages, such as Haskell. If we can package our programs as software components, then it is easy to integrate them into applications written in other languages.

In earlier work we described a preliminary integration of Haskell with Microsoft's Component Object Model (COM), focusing on how Haskell can create and invoke COM objects. This paper develops that work, concentrating on the mechanisms that support externally-callable Haskell functions, and the encapsulation of Haskell programs as COM objects.

1 Introduction

"Component-based programming" is all the rage. It has come to mean an approach to software construction in which a program is an assembly of software components, perhaps written in different languages, glued together by some common substrate [16]. The most widely used substrates are Microsoft's Component Object Model (COM), and the Common Object Request Broker Architecture (CORBA). The language-neutral nature of these architectures offers a tremendous new opportunity to those interested in exotic languages such as Haskell (our own interest): if we can present our programs in COM or CORBA clothing, then the client programs will neither know nor care that the program is written in Haskell. Our Haskell programs can thereby inter-operate with a huge variety of other software, and a would-be user of Haskell is not faced with an all-or-nothing choice.

In an earlier paper we described how to instantiate and invoke COM objects from a Haskell program [11]. In that paper we implied that it would be but a short step to be able to seal up a Haskell program inside a COM object, thus completing the picture. In practice, this ability proved more subtle than we had supposed. This paper tells the story.

ICFP '99 9/99 Paris, France

The main contribution is the overall *design* of our Haskell COM server. More specifically:

- Our design is carefully factored, so that it can easily work with a variety of Haskell implementations, including interpreters (the latter is trickier than it may at first appear). Most of the required functionality is encapsulated in our separate H/Direct tool, or in library modules written in Haskell. This "arms-length" design does not come at the price of convenience; it is still extremely easy to create COM components, and to implement a COM component in Haskell. Many other COM interfaces have a tighter, and hence less portable, integration with the compiler (Visual Java, for example).
- The only facility required from the Haskell implementation is a foreign language interface that (a) supports the import and export of Haskell functions, and (b) provides hooks for managing pointers from Haskell to the external world, and back again. Our earlier paper described `foreign import` and `foreign export`, extensions to Haskell that allow it to call, and be called by, an external program. It turned out that to support callbacks and COM objects we need a more dynamic form of these primitives, `foreign import dynamic` and `foreign export dynamic`. We motivate and describe these primitives (Section 3).
- Even though COM does not support parametric polymorphism, we show how polymorphism can be used to: encode the (interface) inheritance structure of interface pointers; connect interface pointers with their globally-unique identifiers (GUIDs); and ensure that object vector tables are only paired with appropriate object states (Section 5).
- COM is very general, but it requires quite a bit of C++ code to build a COM object, usually supported by "wizards" of some sort. We are instead able to provide a library of higher-order functions that make it easy to construct COM objects without wizardly support (Section 6).

Overall, we give an elegant and easy-to-use design for using building and using COM objects in Haskell. In some ways there is nothing really difficult about it, but it has nevertheless taken us over a year to evolve, so it is certainly a more subtle task than we initially appreciated.

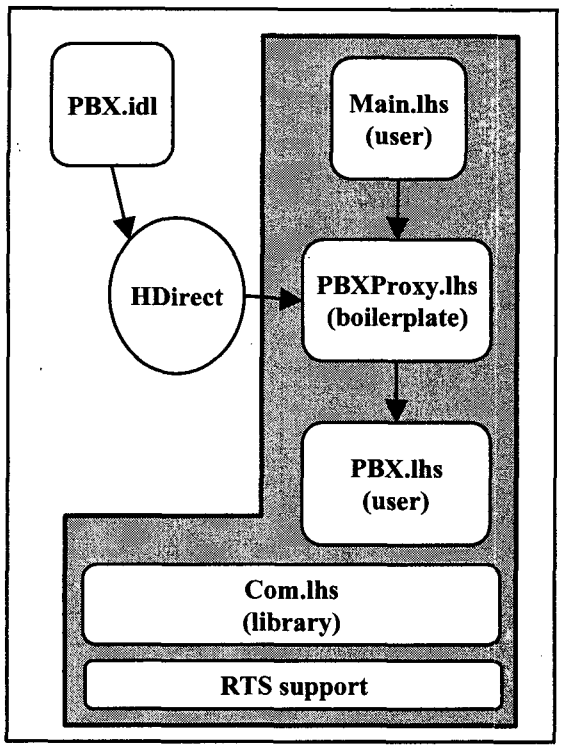

Figure 1: A COM component in Haskell

2 Overview

We begin by giving an overview of our architecture (Figure 1). A Haskell program (grey box) that implements a COM component consists of three parts:

- The application code, written in Haskell by the programmer (labelled "user" in Figure 1).
- A collection of automatically-generated Haskell "boilerplate modules", one per COM class. Each of these modules is generated by our H/Direct tool from an Interface Definition Language (IDL) specification of the class.[1] These modules deal with the "impedance mismatch" between Haskell and COM.
- A Haskell library module, `Com`, which exports all the functions needed to support COM objects in Haskell (labelled "library"); and a C library module that provides some run-time support.

Our earlier paper discusses the pros and cons of using a separate interface definition language, IDL, to define the interface between COM components, and we do not repeat that discussion here [2]. Notice, however, that we use IDL and H/Direct *both* when invoking a COM object from a Haskell program, *and* when implementing a COM object in Haskell. In each case data flows across the border in both directions, so there are clear similarities.

Notice that H/Direct generates only Haskell modules; it does not also generate C code. This design choice minimise the number of files and tools that the programmer has to deal with.

[1] Optionally, the boilerplate code can be put inside one module.

3 The foreign function interface

H/Direct generates Haskell code that marshalls values between Haskell and the foreign language. But in the end, it must generate a real call to the foreign procedure, passing parameters. This foreign call can only be expressed using some extension to the Haskell language. The same is true if we want a foreign procedure to call a Haskell function. In this section we describe a set of language extensions that address this need.

We have carefully minimised what is required from the language implementation, while maximising the work done by H/Direct. In this way, any Haskell that implements our extensions can interface with COM, using the implementation-independent H/Direct to do most of the work.

3.1 Foreign static import and export

Earlier versions of GHC (the Glasgow Haskell Compiler) provided `ccall` (or even `casm`) to invoke a C procedure [12]. However, while this facility is (fairly) easy to support in a compiler that uses C as an intermediate language, it is a bit more difficult when using a native code generator, and well-nigh impossible when using an interpreter such as Hugs. Furthermore, it says nothing about how to allow C to call Haskell, or how to inter-operate with procedures with non-C calling conventions.

Our new foreign function interface is much simpler. Here is an example of how to import a foreign procedure:

```
foreign import "hash32" hash :: Int -> IO Int
```

This `foreign` declaration is modeled directly on the `primitive` declaration that Hugs has supported for some time. The declaration defines the Haskell `IO` action `hash` which, when invoked, will call the external procedure `hash32`. The implementation of `hash` also takes care of converting between the Haskell representation of an `Int` and the corresponding external representation.

The result of `hash` has type `IO Int` rather than simply `Int`, to signal that `hash` might perform some input/output or have some other side effect. We give a short summary of the `IO` monad in the Appendix.

The range of types that can be passed to and from a foreign-imported procedure is deliberately restricted to the (small) set of primitive types. By a "primitive type" we mean one that cannot be defined in Haskell, such as `Int`, `Float`, `Char`. Only the language implementation knows the representation of primitive types, and so only the language implementation can marshall them. For all other types, such as lists or `Bool`, H/Direct is used to generate marshalling code. The same restriction applies to the other variants of `foreign` that we discuss later, for the same reasons.

3.2 Variations on the theme

We support several variants of the basic `foreign` declaration:

- The name of the external procedure can be omitted, in which case it defaults to the same as the Haskell procedure.

```
foreign import hash :: Int -> IO Int
```

- If the programmer is sure that the foreign procedure is really a function — that is, it has no side effects — he can write the type as a non-IO type:

```
foreign unsafe import "sin"
  sin :: Double -> Double
```

  The "unsafe" keyword highlights the fact that the programmer undertakes a proof obligation, namely that the function really is a function. We use this convention uniformly (also e.g. in `unsafePerformIO`), so that a programmer can find all his proof obligations by saying `grep unsafe`.

- By default, `foreign import` uses the C calling convention, but the convention can instead be specified explicitly:

```
foreign unsafe import ccall "sin"
  sin :: Double -> Double
```

  We also support the standard calling convention (`stdcall`) used in Win32 environments.

- In many systems it is necessary to specify the library or DLL[2] in which the external procedure can be found.

```
foreign unsafe import "MathLib" "sin"
  sin :: Double -> Double
```

A similar declaration allows the programmer to expose a Haskell function to the outside world:

```
foreign export "put_char" putChar :: Char -> IO ()
```

This exports a C-callable procedure `put_char` that in turn invokes the Haskell function `putChar`, marshalling the parameter appropriately. The calling convention can be specified, just as with `foreign import`, and a pure (non-I/O) Haskell function can be exported just as easily (no need for "unsafe" here):

```
foreign export fibonacci :: Int -> Int
```

Similar to the foreign import case, when the external name of the exposed function is ommited, it defaults to the same name as the Haskell function.

3.3 Stable pointers and foreign objects

It is often necessary to pass a Haskell value (pointer) to an external procedure. This raises two difficulties: first, the Haskell garbage collector cannot tell when the Haskell value is no longer required; and second, the value may be moved by the (copying) garbage collector. We solve both these problems by registering the Haskell value as a *stable pointer*. This registration (a) returns a stable value (a small integer) that names the value, and will not change during garbage collection, and (b) tells the garbage collector to retain the value until told otherwise. Subsequently, the stable pointer can be dereferenced to recover the original Haskell value.

An exactly dual problem arises when we want to pass to a Haskell program a pointer to an external object (e.g. a file handle, `malloc`'d block, or COM interface pointer). Often, we would like to be able to call `fclose`, or `free`, on the external reference when the Haskell garbage collector finds that it is no longer required. Such "run this when the object dies" behaviour is called *finalization*.

We have defined extensions to Haskell to support both stable pointers and finalisation. They are described in detail in a companion paper [10], so we do not discuss them further here.

3.4 Dynamic import

The `foreign import` primitive is fine if we know the name of the C function we want to invoke. But sometimes we don't. Notably, when invoking a COM object, we start from an *interface pointer*, which points to a location that points to a vector table of methods (we discuss this more in Section 4). To invoke the method, we must fetch the address of the method from the vector table, and call it. `foreign import` simply doesn't do the job; it works fine for link-time or load-time binding, but not at all for run-time binding.

To address this deficiency, we first need a new primitive Haskell data type, `Addr`, that represents a machine address. (We could have used `Int`, but that seems unsavory.) Next, we extend `foreign import` with a `dynamic` attribute:

```
foreign import dynamic
  hashMethod :: Addr -> (Int -> IO Int)
```

This defines a Haskell function `hashMethod`, whose type is as specified. Function `hashMethod` takes the address of the foreign procedure, which must be of type `Addr`, and returns a fully-fledged Haskell function that, when applied, will invoke the foreign procedure. Consider the following example:

```
do{ h  <- ...get addr of hash procedure...
       -- h has type Addr
  ; let hash = hashMethod h
  ; r1 <- hash 34
  ; r2 <- hash 39
  ; ...
  }
```

`h` is the address of a suitable C procedure; `hashMethod` turns `h` into a Haskell function of type `Int -> IO Int`, which is then invoked twice. Of course, if `h` is bound to a bogus address then terrible things will happen.

It is rather simple to implement `foreign import dynamic`. The only difference from the static version is that the call takes place to a supplied argument, rather than to a static label. This contrasts sharply with its dual, dynamic export, which we study next.

3.5 Dynamic export

Just as `foreign import` is inadequate in general, so is `foreign export`, for two reasons. First, `foreign export` only makes sense in a compiled setting, since its effect is to generate a code label that is externally visible; an interpreter cannot reasonably implement `foreign export`.

[2]Dynamically Linked Library

Second, `foreign export` works on *top-level* functions. But we might want to export arbitrary functions. For example, external library procedures quite often take a *callback* parameter; that is, a pointer to a procedure that the external procedure will itself call. For example, the Win32 API provides a function that allows you to iterate over the current list of open windows:

```
typedef BOOL (*WNDENUMPROC)(HWND, LPARAM);
BOOL EnumWindows( WNDENUMPROC enumFunc
                , LPARAM lParam
                );
```

The system call takes a pointer to a callback procedure to invoke for each open window, together with a value `lparam` that we'll ignore for now. The callback procedure returns a boolean value to indicate whether we should stop iterating over the windows or not.

The system call itself can easily enough be imported into Haskell[3]

```
type BOOL    = Int
type LPARAM = Int
type WNDENUMPROC = Addr

foreign import "EnumWindows"
  enumWindows :: WNDENUMPROC -> LPARAM -> IO BOOL
```

But what to do with the callback? We want to implement it in Haskell, so the callback will have to be dressed up to appear like a C function pointer. One way would be to use `foreign export` to export a Haskell procedure as a C procedure, and add some mechanism to give Haskell access to the address of that C procedure, to pass to `enumWindows`.

But there is a much more elegant solution. We provide a `dynamic` form of `foreign export`, thus:

```
type HWND   = Addr
foreign export dynamic
   mkWndEnumProc :: (HWND -> LPARAM -> IO BOOL)
                 -> IO WNDENUMPROC
```

This declaration defines a Haskell function `mkWndEnumProc`, with the type specified. Function `mkWndEnumProc` takes an *arbitrary Haskell function value* of the given type as its single argument, and returns a C function pointer. This C function expects to find two arguments on the C stack; it marshalls them into the Haskell world, and passes them to the Haskell function that was passed to `mkWndEnumProc`. Here is an example of its use[4]:

```
windowTitles :: IO [String]
windowTitles =
  do{ ref <- newIORef []
    ; let getTitle :: HWND -> LPARAM -> IO BOOL
          getTitle hwnd lp =
            do{ t  <- getWindowTitle hwnd
              ; ts <- readIORef ref
              ; writeIORef ref (t:ts)
              ; return (boolToInt True)
              }
    ; cback <- mkWndEnumProc getTitle
    ; enumWindows cback (0::Int)
    ; readIORef ref
    }
```

Here, `getTitle` is the callback procedure; it is called for each window, passing the window handle and the `LPARAM` value. It in turn calls `getWindowTitle` (another foreign-imported procedure) to get the window title, and puts it onto the front of a list of window titles, kept in a Haskell mutable variable `ref`.

The *Haskell* function `getTitle` is turned into a *C-callable* procedure `cback` (of type `Addr`) by `mkWndEnumProc`, the function defined by the `foreign export dynamic` declaration. Finally `cback` is passed to `enumWindows`.

Phew! We do not want to claim that this is beautiful programming style. For example, it is rather gruesome to use a mutable variable in `getTitle`. But the style is dictated by the architecture of Windows system calls; we are stuck with it. However, we are now ready to understand quite a bit about `foreign export dynamic`:

- The callback function `getTitle` is a first class Haskell value. It is not a top-level function, as must be the case for a static `foreign export`. In this case, `getTitle` has a free variable, `ref`, the mutable cell that it updates.

  This capability is modeled in C by the `LPARAM` parameter. The system call accepts `LPARAM` as well as the callback procedure, and passes `LPARAM` each time it calls the procedure. In effect, the (callback, `LPARAM`) pair constitutes a closure, of code plus environment.

  In this particular case, a C programmer would use `LPARAM` to point to a location in which the list is accumulated, just like `ref`. If there were many free variables, matters would be less simple. The Haskell programmer does not need to bother with `LPARAM` — indeed, `lp` is unused in the definition of `getTitle`. `mkWndEnumProc` captures a first-class Haskell value, free variables and all. Higher-order programming in C!

- `mkWndEnumProc` encapsulates a Haskell value as a C function pointer. To do this, it first registers the Haskell value as a stable pointer (Section 3.3), and then embeds the stable pointer in the C function. The programmer can explicitly free the retained Haskell value using:

  ```
  freeHaskellFunctionPtr :: Addr -> IO ()
  ```

  This operation cannot be done automatically, since it depends on knowing that the exported function pointer is no longer needed externally.

- As with the other `foreign` declaration variants, a `foreign export dynamic` also allows you to specify which calling convention the returned function pointer should expect.

3.6 Implementing dynamic export

Dynamic export is considerably harder to implement than dynamic import, because we have to generate a C function pointer *that cannot be static*, because it must somehow refer to the Haskell function it encapsulates. This forces us to perform a little bit of dynamic code generation.

Our implementation for the Glasgow Haskell Compiler works by taking advantage of the *static* version of `foreign`

[3] We declare types `BOOL`, `LPARAM`, etc as Haskell type synonyms that mimic the C header file definitions of these types. Such type declarations are usually generated automatically by H/Direct.

[4] The Appendix introduces `IORefs`.

export. Here, for example, is how we implement mkWndEnumProc. We repeat its declaration here:

```
foreign export dynamic
  mkWndEnumProc :: (HWND -> LPARAM -> IO BOOL)
                -> IO WNDENUMPROC
```

GHC first generates code exactly as if the programmer had written:

```
foreign export
  wndEnumProc :: HWND -> LPARAM
              -> StablePtr (HWND->LPARAM->IO BOOL)
              -> IO BOOL
wndEnumProc h l sp =
  do{ f <- deRefStablePtr sp
    ; f h l
    }
```

wndEnumProc takes an extra argument, a stable pointer to the function value (Section 3.3); it simply dereferences the stable pointer, and calls the function it gets back. Now, GHC generates code for mkWndEnumProc, which does three things:

- registers the Haskell function as a stable pointer;
- dynamically generates a code fragment;
- returns the address of this dynamically generated code.

The dynamically-generated code consists of two or three instructions:

```
add-param <function pointer>
jump wndEnumProc
```

The add-param "instruction" must be whatever machine code is necessary to pass one extra parameter — often this is just a matter of pushing it on the stack (perhaps also moving the return address). Once this is done, the statically-exported wndEnumProc will do the rest. Clearly, the dynamic-code-generation part is highly architecture dependent, but it is also very short, and is not hard in practice.

Unfortunately, this solution won't work at all for the Hugs interpreter, because an interpreter can't support static foreign export. Instead, the Hugs implementation of mkWndEnumProc dynamically generates the following segment of machine code:

```
push <function pointer>
push <type descriptor>
jump GenericCaller
```

Here <type descriptor> is a (pointer to a C-format) string that encodes the type signature of the function. The <function pointer> is a stable pointer to the Haskell function value, as before. Finally, GenericCaller is a fixed piece of code that (a) uses the type descriptor to marshall data from C to Haskell, (b) calls the specified Haskell function, (c) marshalls the Haskell result back, and (d) returns to the C caller. GenericCaller is highly machine dependent, since it must know all about the caller's calling conventions; but at least it need only be written once.

3.7 Related work

Foreign function interfaces (FFIs) are clearly of great use, but papers describing them are relatively thin on the ground. Most functional programming systems provide a FFI, allowing calls to external functions to embedded within functional code. However, few provide equally good support for the outside to call in. The esh Scheme implementation [14] is a notable exception; it was designed with the explicit goal of making hybrid Scheme and C/C++ applications easier to write. Another, more recent system is the Bigloo Scheme compiler [15].

For ML-based languages, the Standard ML of New Jersey compiler's foreign function interface does also provide support for call-ins [5]. Function closures can be dressed up behind a C function pointer, which can then be passed out to the outside world, making it similar in power to foreign export dynamic.

A similar approach is provided by the Objective Caml FFI [8], which requires exported functions to be registered by giving them a name (an arbitrary string) from within OCaml code. The run-time system provides a C callable entry point for looking up the OCaml function closure that hides behind a name, and invoke through a class of invocation functions. This scheme requires that the user makes up the difference using C, writing a little bit of stub code that does the lookup and invokes the function by marshalling and unmarshalling the arguments and results. Contrast this with foreign export dynamic which makes the Haskell-nature of the function pointers it returns transparent to the user.

To our knowledge, the only other Haskell system that provides support for externally-callable Haskell functions is the NHC 1.3 compiler [17], which provides a basic export mechanism similar to that of Objective Caml's.

4 How COM works

Before we can describe how to encapsulate a Haskell program as a COM component, we have to digress briefly to explain how COM works. We concentrate exclusively on *how* COM works, rather on *why* it works that way; the COM literature deals with the latter topic in detail [13]. This section is closely based on our description in [11].

Here is how a client, written in C, might create and invoke a COM object:

```
/* Create the object */
err_code = CoCreateInstance ( cls_id
                            , iface_id
                            , &iptr
                            );
if (not SUCCEEDED(err_code)) {
 ...error recovery...
}

/* Invoke a method */
(*iptr)[3]( iptr, x, y, z );
```

The procedure CoCreateInstance is best thought of as an operating system procedure. (In real life, it takes more parameters than those given above, but they are unimportant here.) Calling CoCreateInstance creates an instance of an object whose *class identifier*, or CLSID, is held in cls_id. The class identifier is a 128-bit *globally unique identifier*, or GUID. Here "globally unique" means that the GUID is a name for the class that will not (ever) be re-used for any other purpose anywhere on the planet. A standard utility

27.3.1 – The Registry

The registry is always located at a *pseudo-index*, whose value is defined by `LUA_REGISTRYINDEX`. A pseudo-index is like an index into the stack, except that its associated value is not in the stack. Most functions in the Lua API that accept indices as arguments also accept pseudo-indices---the exceptions being those functions that manipulate the stack itself, such as `lua_remove` and `lua_insert`. For instance, to get a value stored with key `"Key"` in the registry, you can use the following code:

```
lua_pushstring(L, "Key");
lua_gettable(L, LUA_REGISTRYINDEX);
```

The registry is a regular Lua table. As such, you can index it with any Lua value but **nil**. However, because all C libraries share the same registry, you must choose with care what values you use as keys, to avoid collisions. A bulletproof method is to use as key the address of a static variable in your code: The C link editor ensures that this key is unique among all libraries. To use this option, you need the function `lua_pushlightuserdata`, which pushes on the Lua stack a value representing a C pointer. The following code shows how to store and retrieve a number from the registry using this method:

```
/* variable with an unique address */
static const char Key = 'k';

/* store a number */
lua_pushlightuserdata(L, (void *)&Key);  /* push address */
lua_pushnumber(L, myNumber);  /* push value */
/* registry[&Key] = myNumber */
lua_settable(L, LUA_REGISTRYINDEX);

/* retrieve a number */
lua_pushlightuserdata(L, (void *)&Key);  /* push address */
lua_gettable(L, LUA_REGISTRYINDEX);  /* retrieve value */
myNumber = lua_tonumber(L, -1);  /* convert to number */
```

We will discuss light userdata in more detail in Section 28.5.

Of course, you can also use strings as keys into the registry, as long as you choose unique names. String keys are particularly useful when you want to allow other independent libraries to access your data, because all they need to know is the key name. For such keys, there is no bulletproof method of choosing names, but there are some good practices, such as avoiding common names and prefixing your names with the library name or something like it. Prefixes like `lua` or `lualib` are not good choices. Another option is to use a *universal unique identifier* (`uuid`), as most systems now have programs to generate such identifiers (e.g., `uuidgen` in Linux). An `uuid` is a 128-bit number (written in hexadecimal to form a string) that is generated by a combination of the host IP address, a time stamp, and a random component, so that it is assuredly different from any other `uuid`.