11 11 TEX LATEX

HaskellDirect User's Manual - version 0.26

Sigbjorn Finne <sof@dcs.gla.ac.uk> November 1999

This is the user's manual for HaskellDirect (HDirect), a foreign function interface framework for Haskell based around the use of an interface definition language (IDL.)

Contents

1 Introduction

This document describes how to use HaskellDirect (HDirect), a foreign function interface framework for Haskell. HDirect is based around the use of an *Interface Definition Language (IDL)* to specify a programming interface in a programming language independent manner. Why is this of use to the Haskell programmer?

- **Creating Haskell bindings to external libraries.** By describing the set of functions exported by some library using IDL, the HDirect IDL compiler takes care of generating the necessary Haskell (low-level) function stubs that lets you access these external functions from within your Haskell code.

  The compiler supports two flavours of IDL for this, DCE IDL syntax (as used by OSF DCE and the Microsoft IDL compiler) and the OMG CORBA IDL.

- **Creating external bindings to Haskell libraries.** Going the other way, IDL can also be used to specify Haskell functions that you want to export to the outside world, e.g., you want to be able to use Haskell code from within your user interface, which is implemented in C, say. HDirect supports this too, generating the necessary proxies that gives the Haskell functions a C compatible calling interface.

- **Creating Haskell client interfaces to COM objects** A special case of the first point, but a quite important one. When the IDL specification represents a set of COM interfaces, HDirect generates special versions of the Haskell stub functions that lets you use COM objects from within Haskell.

  HaskellDirect supports the same IDL syntax as the Microsoft IDL compiler (MIDL) for describing COM components, but it generates a mapping for Haskell rather than C/C++.

- **Creating Haskell COM objects.** A dual to the above, where HDirect generates the wrappers which lets you implement COM components in Haskell.

This document introduces the various features of IDL and what it corresponds to in Haskell by a series of examples, covering the different modes of operation just outlined. An exact definition of IDL is not presented, instead we introduce the features you're most likely to need by way of examples, but give instead pointers to IDL specifications and other resources at the end.

But, before presenting examples of IDL can be made use of, let's start off by motivating the need for IDL and HaskellDirect a little.

2 Foreign function interfaces and Haskell

Most of the world's software is not written in Haskell. Not earth shattering news this, so how can we 'improve' on this situation? There's (at least) two alternative routes that can be followed to make a language like Haskell a viable alternative when picking the language(s) to use for a project:

- Rewrite all your software + libraries in Haskell, thereby doing it Right.
- Artfully re-use other people's code written in some other language, (and thereby letting your Haskell end-product take most of the credit :)

The first route is a very long one, and it often turns out that reverse engineering a library in Haskell doesn't radically improve on the original programming interface. The Haskell variant of the "not-invented-here" syndrome.

The second option makes much more sense, re-using external libraries from Haskell, customising their programming interfaces to provide a veneer that is more appealing to the Haskell Programmer In The Street.

Support for the Wise Choice is provided by most Haskell systems through their *foreign function interfaces*. One early example of such a beast was (and still is) GHC's support for embedding calls to C functions inside `IO` code with the `_ccall_` construct. Together with built-in compiler support for converting from a primitive set of Haskell types to their expected external representation (and back), this gives the Haskell programmer the ability to interface to external libraries without too much bother. Examples of libraries/applications that have been built using `_ccall_` include interfaces to windowing systems/toolkits (e.g., Haggis, Budgets, etc.), system libraries (e.g., POSIX lib), etc. Other Haskell systems provide similar foreign function interface support (e.g., Hugs and Gofer's `primitive` declarations.)

This is all good stuff and a move in generally the right direction, but current foreign function interfaces fall short in couple of departments:

- **Not portable.** Haskell code using `_ccall_` is inherently GHC-specific, so compiling such code with another compiler breaks. Having a common FFI language/mechanism would be a Good Thing from a user's point of view.
- **Bandwidth problems.** The set of types that can be passed to and from a `_ccall_` is restricted to the set of types supported by C. This means that for Haskell data types that are not part of this set have to be explicitly packed and unpacked to one of the 'C-callable' types by the programmer prior to calling out using `_ccall_`. Writing this out is tedious and error prone.
- **Unidirectional.** Calling out is much better supported than calling in, i.e., it is much easier to call an external function from the Haskell side rather than calling a Haskell function from the outside. In the case of GHC, the latter has only been possible in a very contorted and limited manner.

Based on the experiences with GHC's `_ccall_` and Hugs' `primitive`, *Green Card* `<http://www.dcs.gla.ac.uk/fp/software/green-card/>` was implemented, a foreign function interface preprocessor for Haskell. It addresses the problem of portability by providing a little language for describing the type signatures of external functions, and how to convert from the Haskell data representation of the external function to the expected representations (and back again.) Given such a specification, the implementation of Green Card takes care of generating the Haskell system specific code for actually calling out.

Green Card has proven to be a very useful addition to the Haskell programmer's toolset, but it too has its problems and limitations:

- Green Card describes external functions by giving them a *Haskell type*. In the cases where the Haskell type is isomorphic to the type of the external function, everything is rosy, e.g.,

  ```
  %fun sin :: Double -> Double
  ```

  where the external function `sin` expects a double as argument, and gives one back. However, suppose the external function returns a pair of results:

  ```
  %fun lookupVal :: String -> IO (Bool, Int)
  ```

  where the external function `lookupVal` returns a lookup status value together with the matching integer (if any.) To write this using Green Card, you'll need to use a combination of inline C code and the Green Card's little language for marshalling values between the C and Haskell worlds:

  ```
  %fun lookupVal :: String -> IO (Bool, Int)
  %call (string key)
  %code int res; BOOL status;
  %     status = lookup(key, &res);
  %result (bool status, int res)
  ```

  setting up the call in C, and taking care of converting from the intermediate C variables holding the results into Haskell result returned.

  Another example is when you want provide a Haskell binding to an external function, but with some of its arguments fixed:

```
%fun createWindow :: String -> IO Window
%call (string title)
%code WINDOW w = CreateWindow(title, 100, 100, 200, 200);
%result (window w)
```

which also requires you to write some inline C code to set this up.

When there isn't an exact match between the desired Haskell signature of an external function and its real signature, the programmer is forced to make up the difference using a combination of inline C and make use of Green Card's little language for describing how data should be (un)packed and combined when going between Haskell and C code.

Using C to do the impedance matching is powerful, but Green Card doesn't give you any other options. Building a domain-specific language with an (increasing) number of features and abstractions is also unfortunate, as we've already got a programming language at hand (Haskell) that we might as well be using for this.

- Marshalling user defined data structures isn't a easy as it could be. For instance, the following external (C) structure:

```
typedef struct tagFoo {
  int x;
  int y;
  double z;
} Foo;
```

requires you to define a corresponding Haskell data type and a *data interface scheme (DIS)* which instructs Green Card how to generate code that maps to/from the two representations of the record type.

- Green Card offers no support for using/writing COM objects. To use Haskell as a glue language for combining together software components, a prerequisite is to be able to interface to a component that complies with the conventions of a component model like COM (or CORBA) Green Card doesn't provide any support for this.

- Unidirectional - Green Card also does not address the problem of how to expose Haskell functions to the outside world.

Instead of extending Green Card to address the above points, we've decided to opt for an off-the-shelf solution to the problem: using an *interface definition language(IDL)* to describe foreign functionality. In the cases where there is a difference between the type signature of the external function, and the type signature that the programmer wants to present to the Haskell user, Haskell is used to make up the difference.

3 Using IDL

An Interface Definition Language(IDL) lets you describe both the type signatures of a function and the data types it takes as arguments in an architecture-independent/programming language neutral way. (i.e., it's a specification language, not a programming language).

Since one of the origins of IDL is from distributed (RPC based) programming, it is perhaps not too surprising that the type system it supports (and its syntax) is closely based on that of C, the system programming language of choice at the time when RPC systems were initially developed.

HaskellDirect currently implements two dialects of IDL: the OSF DCE dialect, including the extensions made by Microsoft to accommodate the expression of COM interfaces using IDL. HaskellDirect has also got a front-end for the dialect of IDL used by the OMG IIOP CORBA specification.

Over the next couple of sections, IDL is introduced by way of example, showing you how the HaskellDirect compiler can be used to create Haskell bindings to libraries and COM components.

3.1 Interfacing to C libraries

This section presents a couple of worked out examples where HaskellDirect is used to create bindings to C libraries. If you do want to experiment with these examples (or maybe simply just use), the `examples/` directory of the distribution contains the code.

3.1.1 A math library binding

As an initial example of IDL and how HaskellDirect can help you to create bindings to external libraries, here's the IDL specification for ANSI C's math library:

```
module Math {

  [pure]int  abs([in]int x);
  [pure]long labs([in]long x);
  [pure]double fabs([in]double x);

  [pure]double ceil([in]double x);
  [pure]double floor([in]double x);
  [pure]double fmod([in]double x,[in]double y);

  [pure]double exp([in]double x);
  [pure]double log([in]double x);
  [pure]double log10([in]double x);

  [pure]double frexp([in]double x,[in,out,ref]int* nptr);
  [pure]double ldexp([in]double x,[in]int n);
  [pure]double modf([in]double x,[in,out,ref]double* nptr);

  [pure]double pow([in]double x,[in]double y);
  [pure]double sqrt([in]double x);

  [pure]int rand(void);
  void srand([in]int seed);

  [pure]double cos([in]double x);
  [pure]double sin([in]double x);
  [pure]double tan([in]double x);

  [pure]double acos([in]double x);
  [pure]double asin([in]double x);
  [pure]double atan([in]double x);
  [pure]double atan2([in]double x,
                     [in]double y);

  [pure]double cosh([in]double x);
  [pure]double sinh([in]double x);
  [pure]double tanh([in]double x);
};
```

The functions that make up the math library are all encapsulated in an IDL `module` declaration, which causes the IDL compiler to generate a Haskell module `Math` containing the stubs that take care of interfacing to the external functions.

The specification is almost identical to the ANSI C function prototypes that can be found in a `math.h` include file, but the IDL function signatures does contain some extra information:

- Prefixed to the type of each function parameter is its directionality attribute, indicating in what direction information passes from the caller to the callee: an `[in]` parameter passes information to the callee, an `[out]` parameter goes the other way, whereas `[in,out]` is (as you've probably guessed by now) bi-directional.
- The custom attribute `[pure]` tells the IDL compiler that an external function is to be considered a pure function, so the Haskell version should be given a functional type rather than being an `IO` action.

By attributing the parameter and result types with these extra attributes, the ambiguities of C function prototypes are made clear. The IDL compiler takes advantage of this information when generating the corresponding Haskell stub functions:

```
module Math where

import qualified Prelude

sin :: Prelude.Double -> Prelude.Double
sin x = ...

srand :: Int.Int32 -> IO ()

modf :: Prelude.Double
     -> Prelude.Double
     -> Prelude.IO (Prelude.Double, Prelude.Double)
modf x nptr = ...
```

(Notice how the generated Haskell code takes care of fully qualifying any imported functions or types it uses, avoiding unnecessary name clashes).

The presence of the `[pure]` attribute on `sin` causes it to be given the functional type `Double -> Double`, whereas the side-effecting `srand` is wrapped up as an IO action. The IDL types `double` and `int` correspond to the Haskell types `Double` and `Int32`, respectively (`Int32` is one of the sized numeric types that are part of the Hugs/GHC extension libraries).

The rules for converting from an IDL function type signature with directional parameters into its corresponding Haskell signature is as expected:

- The `[in]` parameters become arguments to the Haskell function (order is preserved).
- The `[out]` parameters together with the function result are returned in the Haskell function's result (all tupled up).
- The `[in,out]` parameters occur in both argument and result position in the corresponding Haskell type.

The HaskellDirect distribution contains the `Math` specification together with some simple code that uses the generated stubs; have a look in the `examples/math` directory.

3.1.2 Fast strings

Here's an example of how to handle character strings more compactly from within Haskell, representing them as C character strings and by providing the standard C string operations over this compact representation. The IDL is as follows:

```
module FastString {

   // [ptr] turns FastString into an abstract type,
   // that is, it will not be packed (or unpacked)
   // into the Haskell representation of a string
   // when passing the border between Haskell and
   // the outside world.
   typedef [ptr]char* FastString;

   [unique]FastString malloc([in]unsigned int sz);
   void strcat([in,out]FastString dest,[in]FastString src);
   void strncat([in,out]FastString dest,
                [in]FastString src,
                [in]int n);
   int strcmp ([in]FastString s1,[in]FastString s2);
   int strncmp([in,string]char* s1,[in,string]char* s2,[in]int n);

   double atof([in]FastString str);
   int    atoi([in]FastString str);
   long   atol([in]FastString str);

   //A White Lie - str*cpy returns the dest. string.
   void strcpy([in,out]FastString dest,[in,string]FastString src);
   void strncpy([in,out]FastString dest,
                [in]FastString src,
                [in]int n);
   FastString strchr([in]FastString s, [in]char c);
   FastString strrchr([in]FastString s, [in]char c);
   int    strpos([in]FastString s, char c);
   int    strrpos([in]FastString s, char c);

   int    strspn([in]FastString s, [in]FastString set);
   int    strcspn([in]FastString s, [in]FastString set);
   FastString strpbrk([in]FastString s, [in]FastString set);
   FastString strrpbrk([in]FastString s, [in]FastString set);

   FastString strtok([in]FastString s, [in]FastString set);
};
```

Points worth noting about this spec:

- To prevent the C strings from being converted into Haskell character strings (and vice versa), the definition uses the `ptr` attribute:

  ```
  typedef [ptr]char* FastString;
  ```

  This attribute signals to the IDL compiler that `FastString` should be treated as a pointer to an external object, and no attempts should be made to unmarshal this object.

- Values of type `FastString` aren't deallocated when the garbage collector has deemed them to be unreachable. This has to be done explicitly, by providing a binding to C's `free()`. In the next section we will see how to use HDirect IDL extensions to shift this memory management burden over on the garbage collector instead.
- As expected, the user of these stubs will have to follow the rules of allocation laid down by the external functions – enough space will have to be allocated for the destination string ahead of call to `strcat`, say. Not particularly convenient to use, but there you go.

Should you want to play with this example, `examples/string` contains the IDL specification along with a simple application that uses it.

3.1.3 The gd library

A third example of how to use IDL to create Haskell bindings to external libraries is an interface to the Gd GIF library (available from *Gd library* `<http://www.boutell.com/gd>`.) The IDL specification for the library is included in the HaskellDirect distribution (see `examples/gd`), but since it's rather long, we won't include it here. Compared to the previous examples, the Gd IDL specification does make use of a couple of features we haven't seen so far:

- The graphics library manipulate an abstract type representing a GIF image. To represent this abstract type in Haskell, the `interface` declaration is used in IDL:

  ```
  interface gdImage {};
  ```

  This informs the IDL compiler that the `gdImage` should be represented as (a pointer to) an abstract type in Haskell. Optionally, the operations provided over the abstract type can be specified within the body of the interface declaration (i.e., within its the curly braces.)

  The name of the generated abstract type will be `GdImage`, i.e., we upper case the first character of the original name to make it a legal Haskell type constructor name.

- To avoid having to explicitly free `GdImage`s when we're through with them, the `interface` declaration is annotated with the `finaliser` attribute:

  ```
  [finaliser("gdFreeImage")]
  interface gdImage {};
  ```

  The attribute forces the abstract type to be represented in Haskell by a *foreign object*. The effect of this is that the garbage collector will now *finalise* `GdImage`s, invoking the finaliser `gdFreeImage`() when the image becomes garbage, i.e., at the point when the garbage collector discovers that the foreign object has become unreachable.

- Another HaskellDirect extension is the `deriving` attribute, which instructs the compiler to add a deriving clause when defining the corresponding Haskell abstract type,

  ```
  [deriving("Eq")]interface FILE {};
  ```

  with the string literal given to the `deriving` attribute is copied verbatim into the generated Haskell source.

- Constants are defined in IDL using the `const` keyword, a feature that's used to define the standard values provided by the Gd header file:

  ```
  const int gdStyled   = -2;
  ```

which will give rise to an equivalent constant on the Haskell side. For better or worse, constant declarations mirror here the `#define`s given in the Gd header file.

The directory `examples/gd` contains the IDL specification along with Haskellised versions of the two demo programs that comes with the gd distribution, `webgif` and `gddemo`.

3.1.4 Strings again

The stub code that the IDL compiler generates take care of converting between the Haskell representation of argument and result values, and their external representations. On top of that, the stub code ensures that the storage taken up by the external data is freed up when we're through with it in the Haskell world. For instance, the following IDL type signature:

```
int strlen([in,string]char* str);
```

generates a Haskell function with the type signature

```
strlen :: String -> IO Int32
```

The body for this (rather pointless) function, marshals the `String` into a zero-terminated sequence of characters before calling the external function. When it returns, the storage allocated for the marshalled string is released.

When the external function returns a value that we want to unmarshal into the Haskell world, the situation changes:

```
[string]char* itoa([in]int i);
```

Here, the result is a zero terminated character string that will be unmarshaled into a Haskell `String`. The type of the generated Haskell stub for `itoa` then becomes:

```
itoa :: Int -> IO String
```

When the `itoa` action has finished unmarhaling the string, it releases the C string. HaskellDirect currently makes the assumption that the C heap is used for all dynamic allocation, so `free()` is used.

In the cases where the character string doesn't point to dynamically allocated memory, but a static chunk, you don't want to call `free()` (or any other freeing routine) when you've finished unmarshaling. To signal that the generated stub code shouldn't free an argument/result, use the custom attribute `nofree`. An example of this attribute in action can be found in `examples/env`.

3.1.5 A zlib binding

As a final example in this section, here's a Haskell binding to `zlib`, a general purpose data compression library (format and methods of compression described in RFC-1950/1951/1952), see *the Zlib home page* `<http://www.cdrom.com/pub/infozip/zlib/>`. The library provides functions for compressing and uncompressing data in memory, and on top of which a `stdio`-like interface to `.gz` compressed files is also provided.

The IDL specification for `zlib` is a relatively straightforward conversion of the contents of its C header file (from the `zlib-1.1.3` distribution), but the spec does use the DCE IDL's support for 'dependent arguments' in a couple of places. For instance, the C function

```
uLong crc32 (uLong crc, const Bytef* buf, uInt len);
```

uses the third argument to carry the length of the sequence of bytes passed as second argument. DCE IDL does provide support for expressing this kind of dependency between arguments,

```
unsigned long crc32  ( unsigned long crc,  [in,size_is(len)]char* buf, [in]unsigned int len);
```

The `size_is()` attribute pinned on the second arg convey this piece of information. The IDL compiler makes use of this when generating the Haskell stub for `crc32`, giving it the type:

```
crc32 :: Word32 -> [Char] -> IO Word32
```

The sequence of bytes is represented as a list, whose length carry the value that the stub will pass as third argument when invoking `crc32()`.

This representation of a buffer of bytes is only meant to illustrate how the `size_is()` attribute is made use of by the compiler; using a more compact buffer representation with `crc32()` is probably worth your while if you care about performance. (A future extension of HaskellDirect would be to generate stubs here that instead of lists uses a more compact representation, such as a byte array.)

The `zlib` specification can be found in `examples/zlib` together with an extremely simple unzipper example.

3.2 Interfacing to Haskell libraries

To demonstrate how HaskellDirect can be used to generate wrappers that lets you package up Haskell code behind a C-callable interface, let's try to expose the following Haskell module to the world:[1]

```
module MathLib where

nfib :: Int -> Int
nfib n | n <= 1    = n
       | otherwise = 1 + nfib (n-1) + nfib (n-2)

fac :: Int -> Int
fac 1    = 1
fac n    = n * fac (n-1)
```

First step, as always, is to define the functionality you want to export using IDL:

```
module MathLib {
 [pure]int nfib([in]int n);
 [pure]int fac([in]int n);
};
```

Use the `[pure]` attribute here to indicate that we're wrapping functions and not IO actions. To generate proxies from the IDL spec, invoke HaskellDirect as follows:

```
sof$ ihc -s -fhs-to-c --gen-headers -fuse-ints-everywhere -c MathLib.idl
```

[1]As I'm sure you'll agree, this is a library that will cause hordes of C developers to come knocking on your door, begging you for a copy :-)

This generates both a Haskell source file, `MathLibProxy.hs`, and a C header file, `MathLib.h`. The generated proxies in `MathLibProxy.hs` takes care of marshalling values between to and from the functions that `MathLib.hs` implements, plus all the FFI details of how to make a Haskell function callable from C.

That's it - to try and build a mixed-language application for yourself, see the `examples/server` directory in the distribution.

3.3 COM client examples

This section contains examples of how to generate and use COM clients using HaskellDirect together with Hugs and/or ghc.

3.3.1 A simple COM client

Suppose we want to use the Internet Explorer Super Label component from within Haskell. To do this, go to the `examples/comcli` directory of the HaskellDirect distribution and type 'make stubs'. This will invoke the IDL compiler and generate the Haskell stubs by reading the type library of the IE Label component (the Makefile has got the path to the component hard-wired in, so you may have to adjust it to make it compile).

- If you want to use HaskellScript or Hugs98, you should now be able to doubleclick on `Main.hs` inside `examples/comcli`, followed by typing `main` at the Hugs command prompt. The result of which should be a nice little message box popping up on your screen.
- If you instead want to use ghc, type `make` inside that directory to build the Haskell Com client. (if that doesn't do it, have a look at the `Makefile` to see if there's anything you'll need to tweak first.)

  When completed, run the binary to (hopefully) see the IE Label About box being displayed on your screen. Congrats - you've successfully built your first Haskell COM client! Not the most impressive of clients, but still :)

3.4 COM component examples

This section contains examples of how to write Haskell COM components, including how to generate the surrounding support code using HaskellDirect.

This currently only works together with ghc.

3.4.1 A telephone directory

All the example code given in this section can be found in the `examples/comserv` directory of the HaskellDirect distribution, together with a `Makefile` for building the component.

We want to write a simple phone directory abstraction which supports lookups and the addition of entries to a directory. The external programming interface to the directory would be as follows:

```
newPBook          :: PBook
addDirectoryEntry :: PBook  -> String -> String -> PBook
lookupName        :: PBool  -> String -> Maybe String
lookupNumber      :: PBook  -> String -> Maybe String
```

What the individual operations do should hopefully be clear - here's one possible implementation of the interface:

```
module PBook
       ( PBook
       , newPBook
       , addEntry
       , lookupNumber
       , lookupName
       ) where

import List ( find )

newtype PBook = PBook [PBookEntry]

data PBookEntry
 = PBookEntry
        { entryName   :: String
        , entryNumber :: String
        }

newPBook :: PBook
newPBook = PBook []

addEntry :: PBook
         -> String
         -> String
         -> PBook
addEntry (PBook ls) name no
  = PBook (PBookEntry{entryName=name, entryNumber=no} : ls)

lookupNumber :: PBook
             -> String
             -> Maybe String
lookupNumber (PBook ls) no
  = fmap (entryName) (find ((==no).entryNumber) ls)

lookupName :: PBook
           -> String
           -> Maybe String
lookupName (PBook ls) name
  = fmap (entryNumber) (find ((==name).entryName) ls)
```

Not much to it, so let's make things a little bit more interesting and try to wrap this phone book up as a COM component.

First step along the way is to try to express the interfaces of a component in IDL, splitting up the operations for looking up entries from that of adding one into separate interfaces:

```
[....]
interface ILookup : IUnknown {
   HRESULT LookupName   ([in]BSTR name,   [out,retval]BSTR* number);
   HRESULT LookupNumber ([in]BSTR number, [out,retval]BSTR* name);
};

[....]
interface IDirectory : IUnknown {
   HRESULT AddEntry ([in]BSTR number, [in]BSTR numb);
```

```
};

[....]
coclass PBook  {
    [default]interface ILookup;
    interface IDirectory;
};
```

The PBook is declared to be a component supporting two interfaces, ILookup and IDirectory. Nice and clean. Unfortunately though, scripting languages such as VBScript and JavaScript forces us to rethink our pursuit of Component Beauty. For one, they only let you access a component via a scripting-friendly IDispatch interface, that is, the notion that a component can consist of more than one interface is just lost on these languages. Only one IDispatch derived interface can be used by these clients, so in order to be more scripting language friendly, let's rephrase the above interface definition:[2]

```
[....]
interface ILookup : IUnknown {
   HRESULT LookupName   ([in]BSTR name,   [out,retval]BSTR* number);
   HRESULT LookupNumber ([in]BSTR number, [out,retval]BSTR* name);
   HRESULT AddEntry     ([in]BSTR number, [in]BSTR numb);
};

[....]
dispinterface DILookup {
  interface ILookup;
};

[....]
library PhBook {

[....]
coclass PBook  {
    [default]dispinterface DILookup;
};
};
```

The dispinterface declaration exposes the methods of ILookup via the aforementioned scripting-friendly IDispatch interface (details of which aren't that important here).

The above specification also embeds the declaration of PBook inside a library declaration. This is done in order for the Haskell implementation of IDispatch, which HaskellDirect generates code for, to work properly.

When operating in 'Haskell COM server' mode, HaskellDirect generates the underlying support code which takes care of the details of how to wrap up a Haskell implementation of a PBook component (see separate document for details of what's generated - ToDo: add xref). The HaskellDirect command line is as follows:

```
ihc -s -c pbx.idl -o PbxProxy.hs --output-tlb=pbx.tlb
```

[2]There are ways and means of working around the problems that IDispatch-based clients such as VBScript poses to the component designer/implementor. An overview of proposed solutions and workarounds can be found at *IDispatch workarounds* <http://www.sellsbrothers.com/tools/multidisp/default.html>. These solutions can be carried over into our setting, but haven't been (yet).

One response to all this might be, "why bother?". The short answer to that is that we dearly want to be able to embed and script Haskell components from within web pages.

where `-s` puts the compiler in server-mode, and `-output-tlb=` is used to generate the *type library* which is used to implement `IDispatch`.

If you feed HaskellDirect the extra option `-skeleton`, it will generate the type signatures and empty definitions of the methods it expects the component implementor to supply. In the case of `PBook`, it generates the following:

```
module PBX where

data State = State

new :: IO State
new = ...

lookupName :: String -> State -> IO String
lookupName name st = ...

lookupNumber :: String -> State -> IO String
lookupNumber number st = ...

addEntry :: String -> String -> State -> IO ()
addEntry name numb st = ...
```

Each method is an IO action that takes as its last argument the state of the `PBook` component which it then updates as needed.

Not quite the signature of our original `PBook` abstraction, but impedance matching the two is not much work:

```
module PBX where

import IOExts
import qualified PBook

data State = State (IORef PBook.PBook)

new :: IO State
new = do
  let x = PBook.newPBook
  r <- newIORef x
  return (State r)

lookupName :: String -> State -> IO String
lookupName name (State st) = do
   pb <- readIORef st
   case PBook.lookupName pb name of
     Nothing -> return "not found"
     Just x  -> return x

lookupNumber :: String -> State -> IO String
lookupNumber number (State st) = do
   pb <- readIORef st
   case PBook.lookupNumber pb number of
     Nothing -> return "not found"
     Just x  -> return x
```

```
addEntry :: String -> String -> State -> IO ()
addEntry name number (State st) = do
   pb <- readIORef st
   let new_pb = PBook.addEntry pb name number
   writeIORef st new_pb
```

Ideally, we shouldn't have to write this layer of code - the HaskellDirect generated code should just 'plug into' the `PBook` signature we wrote earlier, right? That would certainly be nice, but our experiences of writing components in Haskell are so far quite limited, so a pattern of how to express components functionally has yet to emerge. But when or if such a pattern steps forward, HaskellDirect should be extended to avoid having to write code such as the above.

That's it really, next step is to assemble the bits that make up the component, see the `examples/comserv/Makefile` for details of how this is done.

Once you've successfully built the component, make sure you register it to the world using

```
sh$ regsvr32 test.dll
sh$
```

With that done, you can now start to use the component - in `examples/comserv` you'll also find an example of how it can be embedded in a web page (and accessed via VBScript) + there's also some sample code which demonstrates how it can be accessed from a Visual Java client.

4 Invoking the compiler

The IDL compiler processes one or more translation units per invocation, producing one or more output files. An invocation of the compiler takes the following form:

```
ihc [argument...]
```

where command-line arguments are either options or file names.

All command-line options begin with `-`. They may *not* be grouped: `-vO` is different from `-v -O`. Options need not precede filenames: e.g., `ihc *.o -o foo`. All options are processed first and then applied to the input files; you cannot, for example, invoke `ihc -c -g Foo.idl -gc Bar.idl` to apply different backends for the output to be generated for the files `Foo.idl` and `Bar.idl`. For conflicting options, e.g., `-c -noC`, we reserve the right to do anything we want. (Usually, the last one applies).

4.1 Help and verbosity options

A good option to start with is the `-help` or `-h` The IDL compiler dumps a long usage message to standard output and then just exits.

The `-v` option makes the compiler *verbose*: it reports its version number and shows (on standard error) various pieces of information as to what's happening.

If you're just interested in the compiler version number, the `-version` option prints out a one-line string containing the requested info.

4.2 Redirecting the compilation output(s)

The IDL compiler's output normally goes into a `.hs` file (and , possibly, a `.c` file), whose name is derived from the name of the input files. The `-o whale` option overrides the default behaviour and re-directs the output to the file `whale` instead.

If you should happen to give more than one input file on the command line, the use of the `-o` doesn't make any sense, so please don't (if you do, it will just be ignored).

On Win32 platforms, the option `-output-tlb=foo.tlb` redirects the type library output to the file `foo.tlb`.

The option `-output-h=foo.h` tells the compiler to output any C header file output to file `foo.h`. The default behaviour is for the name of the header file to be derived from the name of the Haskell output file.

The option `-odir <dir>` prepends the directory path `dir` to all output filenames.

Notice that if you invoke the compiler as follows:

```
exa$ ihc path/to/my.idl
```

HaskellDirect will output the generated files in the directory `path/to`. Should you want to put them in the current working directory instead, use `"-odir ."`.

The option `-output-module=<name>` can be used to change the name of the Haskell module. For instance,

```
exa$ ihc --hugs path/to/my.idl -o myHugs.hs --output-module=My
```

the file `myHugs.hs`, but instead of deriving the name of the Haskell module from the output filename (`MyHugs`), `My` is used instead.

4.3 Finding imports

An IDL specification may contain one or more `import`, `importlib` or `#include` declarations, instructing the compiler to slurp in information from some other file.

The compiler looks for the file to include along a builtin list of directories (notably including `.`).

`-i<dirs>` **or** `-include-dir <dirs>`

This option prepends a colon separated list of `dirs` to the default import search path.

`-I<dirs>`

In case you're initially washing the IDL specification by running it through the CPP (i.e., using the `-cpp` option), the `-I` option prepends the list of include search path CPP will look in when it encounters a `#include` statement.

4.4 Picking the right front end

The IDL compiler is capable of accepting a couple of IDL variants as input, with the following options controlling which one it should expect:

`-fdce`

Tell the compiler to accept OSF's DCE IDL syntax. The conformance to DCE IDL isn't alltogether strict, in that the couple of minor extensions that HaskellDirect makes will still be recognised.

`-fcom`

Instruct the IDL compiler to accept the language used by the Microsoft IDL compiler, a superset of DCE IDL (extended to support the description of COM interfaces.)

This option also has ramifications for what kind of stub code is generated, see next section.

`-fomg`

This option enables a front-end that accepts the OMG CORBA IDL syntax instead of DCE IDL. This front-end is currently experimental, but should be of some use if you intend to use HaskellDirect and OMG IDL to describe bindings to C libraries (see below).

Please notice that HaskellDirect does not currently ship with a backend and support library that allows you to interface to any CORBA-compliant ORBs.

`-cpp`

Instruct the compiler to initially run the C preprocessor over the input. See Section 4.9 (Running a pre-processor) for details.

The default is to accept the IDL that the Microsoft IDL compiler recognises, i.e., `-fcom`.

4.5 Picking the right back end

What kind of output should be produced is also controllable through a couple of options.

`-fhs-to-c`

The `-fhs-to-c` instructs the compiler that the IDL input should be interpreted as specifying a C library that we want to generate a Haskell binding for. That is, generate stubs which allows Haskell code to call C (hence the name :-)

`-fcom`

The option causes the compiler to interpret the methods in a `interface` declaration to invoke the method via its "interface pointer" (see below for options that control how interface pointers themselves are to be represented.)

If the interfaces are "Automation compatible", that is, they support the `IDispatch` COM interface, the generated stubs can also be used directly with the Automation-friendly *HaskellScript* `<http://www.haskell.org/active/activehaskell.html>`.

`-fautomation`

A close cousin of `-fcom`, which allows you to specify a collection of command-line options that are useful for generating "Automation compatible" stubs. The 'primitive' options it expands to are:

```
-fno-export-list
-fcom
-fcoalesce-methods
-fno-gen-binary-interface
-fenum-instances
-fno-gen-enum-marshallers
```

`-hugs`

Instructs the compiler to generate Haskell stubs that can be used by versions of Hugs that 'just' support `primitive` declarations (and not the new GHC/Hugs FFI). With this option on, a `.c` file is also generated, containing the stubs which perform the external calls, see Section 5.4 (Compiling Hugs stubs) for how to compile it up.

`-gc`

To perform the actual external calls, the HaskellDirect generated stubs uses by default the Hugs/GHC *FFI* <http://www.dcs.gla.ac.uk/fp/software/hdirect>. If, for whatever reasons, you want to use Green Card to generate the code that performs the external calls, the `-gc` option is your friend.

`-g`

Generate Glasgow Haskell specific code that performs the external calls (i.e., `_ccall_` and `_casm_s`). Don't use this option unless you absolutely have to, since the new FFI is really a lot more expressive (and where the future is at.)

`-gen-headers`

Convert the IDL fed as input into C declarations and emit into a C header file.

If used in conjunction with the `-fshort-header`, the generated header file output is dumped in "short form", that is, interface declarations are not expanded, so from

```
interface IA { m1 ; m2 }
```

you only get

```
typedef struct IA* IA;
```

emitted into the header file. This is just what you want when using the generated header file in order to have the generated C stubs compile correctly.

`-gen-tlb`

Instruct the compiler to emit a type library for every `library` declaration it encounters in its input. Type libraries can be viewed as a binary representation of IDL, and are used by OLE Automation clients and servers.

This option is only supported on Win32 platforms.

`-skeleton`

When operating in Haskell component mode, emit a skeleton implementation of the component's methods. For more information on what this fills in, see Section 3.4 (Haskell servers)

`-gen-defs`

A Win32 specific option, which causes the IDL compiler to emit a module definition file (with the suffix `.def`) for each of the DLL you're trying to interface to. This option is of some use when you're interfacing to 'hostile' DLLs that don't come with separate import libraries. By generating the `.def` files for you, you can then easily build the import libraries using standard tools (see Win32 documentation for details on how to do this).

`-h1.4`

The default is for the IDL compiler to emit Haskell code that comply, where possible, with the Haskell 98 standard. If you're working with a Haskell 1.4 system, the `-h1.4` option comes in very useful.

`-noC`

Don't write out the generated stubs. Sometimes useful when debugging.

4.6 Tweaking the output

Like any self-respecting compiler ;-), the IDL compiler supports a fair number of options for tweaking the output generated:

`-include-header=<file>`

When compiling HaskellDirect-generated code, a C compiler is normally involved. For the C stubs generated for Hugs, that's definitely the case, and a C compiler is also involved when using GHC (native code generators aren't quite there.)

In the case of Hugs C stubs (see Section 5.4 (Compiling Hugs stubs) for more details), prototypes and type definitions specific to the API will need to be in scope for the C compiler to operate. For GHC compiled code, you may also need to do this (i.e., you're not using GHC's `-femit-extern-decls` option) to have the C compiler use the right calling convention.

To have a header file be included in either scenario, use `-include-header="foo.h"` or the HaskellDirect-supported IDL extension `stub_include`:

```
stub_include <foo.h>;
stub_include "bar.h";
```

`-include-c-header=<file>`

Closely related to the previous option, but `-include-c-header=<file>` only include `file` in any generated C stubs (including generated header files.)

`-fcompress-enums`

IDL `enum` declarations lets you provide a mapping between symbolic name and an `int` tag values. Mapping these into Haskell requires the generation of a Haskell enumeration data type along with an `Enum` instance for the type.

By default, HaskellDirect will spot whether it can make use of Haskell's `deriving` declaration to have the Haskell system automatically generate the `Enum` instance, i.e., if the enum's tag sequence is of the form `[0,1,2. ...]`.

By supplying the option `-fcompress-enums`, HaskellDirect will try to spot other common progressions and generate `GHC-specific Enum` instances which maps to and from tag values and their Haskell equivalent data constructor tags without wasting lots of code space.

`-fone-mod-per-iface`

This option causes the generated stubs to be split up into one module per interface encountered in the input.

`-fno-export-list`

For each Haskell module generated, don't generate an export list. The effect of this option is to export everything.

`-fexport-with-tysig`

If generated, augment the export list with the types of the functions being exported. Sometimes a useful way to document what's on offer from a Haskell module.

`-fexport-abstractly`

The default is to export the types in the generated Haskell module, non-abstractly, i.e., the data constructors can be seen (and used) outside of the module boundaries. To turn this default behaviour off, use `-fexport-abstractly`.

There's currently no way of specifying on a per IDL type basis whether its Haskell type should be exported or not - just all or nothing.

`-fno-module-header`

Leave out the Haskell module header entirely. You may want to do this if you supply your own using an `hs_quote()` declaration.

`-fno-import-lists`

Generate Haskell import declarations, but don't include list of *exactly* what is being imported. The default behaviour is to include the import lists.

In case you're processing large IDL specifications, one way to cut down the compilation time is to use `-fno-import-lists`, since it saves having to make a separate pass over the Haskell code (about to be) generated.

`-fno-imports`

Don't generate any import declarations at all – let the user supply them separately.

`-fno-qualified-names`

The default is for the generated Haskell source to refer to the imported types and values by their *qualified* names, so as to eliminate any potential name clashes. The `-fno-qualified-names` option disables this behaviour.

`-fno-share-ffi-decls`

By default, HaskellDirect tries to re-use `foreign import` declarations within a module. If two methods/functions map down to a `foreign import` declaration with the same type (and calling convention), only one declaration will be generated.

In some cases, this significantly reduces the number of FFI decls that need to be generated, but should you want to turn off this 'smart', use `-fno-share-ffi-decls`.

`-fout-pointers-are-not-refs`

OSF DCE IDL (and Microsoft's implementation of it) has the restriction that `[out]` parameters have to be references (at the toplevel). `-fout-pointers-are-not-refs` relaxes this (sensible) restriction, and is useful if you want to return a result but not unmarshall it into Haskell. Having `[out]` parameters return references is still possible with this option turned on, but you now have to be explicit about it.

`-fuse-ints-everywhere`

Don't try to make use of the sized integer types that the Hugs/GHC interfaces `Int` and `Word` provide. Instead, just stick with good old `Int` throughout.

`-fint-is-int`

By default the mapping for the IDL type `int` is `Int32` (on 32-bit platforms). Use `-fint-is-int` to have it map to `Int` instead.

`-fint-prelude-coercions`

Up until recently, both GHC and Hugs exported the functions `toInt` and `fromInt` via the Prelude. GHC has with 4.04 stopped doing so, providing these two now via the `Int` interface instead. HaskellDirect-generated code will by default assume that `Int` is where these guys live. (the Sept99 release of Hugs98 exports them through `Int` as well as through the Prelude.)

Should your Haskell system insist that you pick `toInt` and `fromInt` from the Prelude instead, use `-fint-prelude-coercions` to achieve the desired effect.

`-fint-as-word`

Oops, use of the previous option will still run the risk of generating uses of `Word`, a ghc specific type representing unsigned `Int`s. To coerce these guys into `Int` as well, use `-fint-as-word`.

Notice that the marshalling library that HaskellDirect supplies makes use of the numeric types provided by the `Int` and `Word` interfaces, so your quest to create 'Ints-everywhere' world view may not be complete with the addition of `-fint-as-word` and `-fuse-ints-everywhere` to your command line...(I guess what I'm trying to say is that these two options haven't been put to extensive use!)

`-fignore-helpstrings`

The default is for the help strings that are supplied with the `helpstring()` attribute to be included in the generated Haskell source next to the method the help comment applies to. Should that annoy you, use `-fignore-helpstrings` to turn it off.

`-fignore-hidden-methods`

Don't generated any code for methods that have the attribute `hidden` pinned onto them.

Note: As an IDL extension, HaskellDirect supports the `ignore` attribute. It has the same effect as the use of `hidden` coupled with the use of `-fignore-hidden-methods`.

`-fignore-restricted-methods`

Don't generated any code for methods that have the attribute `restricted` pinned onto them.

`-fno-qualified-instance-methods`

If qualified names are to be used in the generated source, turn off the use of qualified names for method in instance declarations. This is required if you're targetting Hugs, but not if you're working with GHC (where it's required that you don't use this option ;-) With respect to the Haskell 98 standard, GHC is the one behaving correctly.

`-fignore-source-interfaces`

Some interfaces are marked as being "source" interfaces, i.e., in the case we're generating client COM stubs, interfaces that the client is supposed to implement (e.g., out-going event interfaces.)

If the client context is either not interested in these interfaces or simply doesn't support them, use `-fignore-source-interfaces` to avoid emitting any code for these.

Notice that HaskellDirect doesn't support the generation of code that lets you implement COM components in Hugs, so uses of `-hugs` implies `-fignore-source-interfaces`.

`-flonglong-as-integer`

Hugs doesn't support fully support the `Int64` type, so if the functions/methods you interface to makes use of `long long`, use `-flonglong-as-integer` to have `long longs` map to `Integer` on the Haskell side.

`-fno-dependent-args`

Experimental option for turning off the special handling of dependent fields/parameters that HaskellDirect performs.

`-fclassic-name-mangling`

By default, IDL names are converted into Haskell type or variable names by changing the casing on the initial letter. For example, `iDL_NAME` goes to `IDL_NAME`, if the name represents a Haskell type/class/module name, but stays as `iDL_NAME` if not. An alternative mapping is provided by the `-fclassic-name-mangling` option, which will transform `iDL_NAME` into either `idlName` or `IdlName`, depending. That is, a sequence of all-upper case letters terminated by an underscore (or end of name) is converted into a capitalised word, and the underscore is removed. Some people like that better.

`-unsafe-calls`

The GHC/Hugs FFI makes the distinction between safe and unsafe calls to the outside world. By default, HaskellDirect generates call-outs of the safe kind, but by using `-unsafe-calls`, it'll do the opposite.

`-fpattern-as-lambda`

The Haskell function definitions generated by the IDL compiler will by default bind function arguments using pattern variables, e.g.,

```
methFoo x y = ...
```

The `-fpattern-as-lambda` option transfers the argument binders to the right hand side,

```
methFoo = \ x \ y -> ...
```

which is compatible with some people's programming style.

`-funwrap-singleton-structs`

Tell the compiler to treat IDL `struct` types with only a single field as being equal to the type of that field.

`-fremove-empty-structs`

If the input contains empty struct declarations, e.g., `"typedef struct foo bar"`, simply ignore these.

4.7 Debugging the output

Should anything go wrong or you simply want to examine intermediate representations of the IDL input, a couple of options for dumping debugging information are provided:

`-d` **or** `-debug`

Enable the output of a little bit of debugging information on stderr.

`-dshow-passes`

Report on stderr what phase of the compiler is currently processing the input.

`-ddump-idl`

Output the (pretty printed) abstract syntax tree on stderr.

`-ddump-ds`

Dump a simplified, and desugared IDL representation on stderr.

`-ddump-rn`

Dump a desugared and renamed IDL representation on stderr. This differs from the desugared output in that IDL names have been transformed to make them unique within the Haskell modules they're about to get mapped to.

`-ddump-absH`

Output the abstract Haskell representation of the input on stderr.

`-output-dump=file`

Instead of outputting the dump on `stderr`, output it to `file`.

4.8 COM specific options

Since one of HaskellDirect's goals is to support the integration of Haskell with COM, it has got some options specific for the COM backend:

`-fno-gen-binary-interface`

When generating stubs that has to work with *HaskellScript* `<http://www.haskell.org/active/activehaskell.html>`, non-Automation interfaces are not of interest, due to the fact that they currently cannot be handled directly. This option causes the compiler to ignore such interfaces, emitting just a message informing the user what's happened.

This option is of less use now that HDirect is capable of generating Hugs stubs to binary interfaces.

`-fuse-dispids`

For Automation compatible interfaces, method invocation is either done by referring to the method by name (the default) or by method id (specified using the `id()` attribute.) Going via method ids (or, dispids, using Automation lingo) is quicker as it saves performing the method name to identifier mapping each time the method is invoked. If you want to do this, `-fuse-dispids` is your friend.

`-fkeep-hresult`

Methods of a COM interface return an `HRESULT` to indicate the success or otherwise of the method invocation. The HaskellDirect generated stubs will by default look at this status code to determine whether to flag an exception or not.

To disable this inspection of the `HRESULT` an instead have it returned back along with the other results of the method, use `-fkeep-hresult`.

`-fcoalesce-methods`

When working with Automation interfaces, you'll sometimes find that methods of different interfaces have the same name and take the same list of parameters (modulo the interface pointer, of course.) The `-fcoalesce-methods` option combines together the stubs for isomorphic methods into one. For example,

```
[...]
interface IA : IDispatch {
   HRESULT lookup([in,string]BSTR name, [out]int* res);
   ...
};

[...]
interface IB : IDispatch {
   HRESULT lookup([in,string]BSTR name, [out]int* res);
   ...
};
```

With `-fcoalesce-methods` enabled, only one `lookup` stub will be generated:

```
lookup :: Int -> IDispatch a -> IO Int
```

Notice that this isn't entirely typesafe, since `lookup` can now be applied to any `IDispatch` interface pointer, not just `IA` and `IB` ones. If you do, a Haskell IO exception will be raised, telling you that method `lookup` isn't supported by that particular object.

In some cases, this trade-off between type safety and size of the generated stub code is worth considering, since method coalescion can drastically reduce the amount of stubs you need to generate for particular components / object models.

`-fsubtyped-interface-pointers`

This option is now enabled by default, so you really shouldn't have to use it any longer. It causes interface pointers to be represented in Haskell in a way that allows us to easily reuse the method stubs of the interface(s) we're inheriting from. To demonstrate, suppose you have the following two interfaces:

```
[...]
interface IPlus : IUnknown {
   HRESULT add([in]int i1, [in]int i2,[out]int* res);
};
[...]
interface IPlusMinus : IPlus {
   HRESULT sub([in]int i1, [in]int i2,[out]int* res);
};
```

i.e., you can perform the `add` operation on an instance of the `IPlus` interface, while instances of `IPlusMinus` also allow you to perform the `sub` operation. Having to generate one `add` stub for `IPlus` and `IPlusMinus` would be tedious in the extreme, so let's avoid this by representing the interface pointers themselves as follows:

```
type IPlusMinus  a = IPlus (PlusMinus a)
data IPlusMinus_ a = IPlusMinus_

type IPlus a  = IUnknown (Plus a)
data IPlus_ a = IPlus_
```

A value of type `IPlus` is a COM interface pointer that, *at least*, supports the methods of the `IPlus` interface. The `IPlusMinus` interface is of `IPlus`'s ilk, being defined as a synonym for `IPlus`. Expanding out the type synonyms is illuminating:

```
type IPlusMinus a = IPlus (IPlusMinus_ a)
                  = IUnknown (IPlus_ (IPlusMinus_ a))
```

i.e., by parameterising the interface pointers over a type that describes the extension of the interface, the relationship between `IPlus` and `IPlusMinus` interfaces is maintained at the Haskell type level, with `IUnknown`'s type parameter envoding the interface inheritance chain.

The dummy types `Plus` and `PlusMinus` are needed to let the type checker do it's job of catching illegal uses of interface pointers, e.g., the application of `sub` to an interface pointer that just implements `IPlus`.

Since `IPlusMinus` is a synonym for `IPlus`, the generated stub for `add` can now also be used by `IPlusMinus`:

```
add :: Int -> Int -> IPlus a -> IO Int
sub :: Int -> Int -> IPlusMinus a -> IO Int
```

`-funparamed-interface-pointers`

Use a representation of interface types that isn't parameterised over the sub type, but a mere type synonym to the interface type it inherits from, e.g.,

```
type IPlus = IUnknown
type IPlusMinus = IPlus
```

`-fignore-dispinterfaces`

For a COM component that support more than one IUnknown interface, there's a problem when it comes to providing an Automation interface to the same functionality, since supporting multiple `IDispatch` interfaces is tricky.

One solution to the problem is to define one big 'dispinterface' containing all the methods of the various binary COM interfaces. Not particularly pretty, but that's how several IDL specs solve this problem.

When generating client bindings, you're not really interested in both the `IDispatch` interface to a method and the 'binary' one. In case it's the latter, `-fignore-dispinterfaces` causes the compiler to simply overlook `dispinterfaces` when generating stubs. Sometimes useful.

The `-fignore-dispinterfaces` option is the inverse of `-fno-gen-binary-interface` option.

`-fsort-defns`

The compiler currently makes the assumption that the definition of an IDL entity preceedes any of its uses (as does MIDL and other IDL compilers). So, if you feed it input where this isn't the case, use -fsort-defns to dependency sort the declarations before processing them further.

With a little bit of work this assumption could be dropped, but until that time, `-fsort-defns` is your friend.

Note: if you're feeding HaskellDirect input you got through the use of Microsoft's OleView utility, `-fsort-defns` is required, as that tool does not sort the IDL it generates. To avoid such troubles alltogether, simply avoid OleView and have HaskellDirect process the type libraries for you directly.

`-fdual-vtbl`

Dual interfaces can either be invoked via the vtbl (i.e., 'binary' invocation) or through `IDispatch` and `Invoke()`. The option `-fdual-vtbl` instructs the compiler to generate stubs that perform method invocation via the vtbl (the default is the other way around.)

`-fno-expand-inherited`

For interfaces that inherit from a non-'builtin' interface (i.e., `IUnknown` and `IDispatch`), don't expand the contents of the inherited interface when compiling the interface that inherits. Experimental, not really of use with the current interface pointer representation.

`-fprefix-interface-name`

Prepend the name of the interface to the names of the Haskell method stubs. This is one way to fight namespace clashes, the default scheme used is to make method names unique within one Haskell module by appending a unique number to methods with the same (Haskell) name, i.e., if there's two methods called `next`, the second will be renamed to `next0`. `-fprefix-interface-name` offers an alternative to this scheme, mapping `next0` to `iEnumFoo_Next` instead.

`-fno-overload-variant`

By default, argument occurrences of the Automation `VARIANT` type will cause HaskellDirect to overload the corresponding Haskell method argument using the `Automation.Variant` type class.

To turn this behaviour off, use `-fno-overload-variant` to have `VARIANT` map to `Automation.VARIANT` instead; a pointer to an externally-allocated `VARIANT` value.

`-fno-enum-magic`

The COM specification specifies the assumed 'shape' of *enumeration interfaces*, i.e., interfaces that lets you enumerate the elements present in a collection. The only difference between two enumeration interfaces is the type of elements that they allows the client to enumerate over. So, providing a shared set of stubs for these enumeration methods makes good sense, parameterising the stubs on how to unpack the enumeration values the interface methods return.

HaskellDirect will by default generate code that makes use of these parameterised enumeration stubs, but should you for one reason or another not want to use these, use `-fno-enum-magic` to turn this default off.

`-fno-mangle-interface-names`

Turn off interface name mangling for interface dummy types. Experimental and not a particularly useful option.

`-tlb`

HaskellDirect supports the reading of type libraries on Win32 platforms. Use `-tlb` to try to open the input files on the command line as type libraries.

`-ftreat-importlibs-as-imports`

A deprecated option, which instead of trying to import type information from a type library when it encounters a `importlib()` statement, will try to import from the corresponding IDL (source) file instead.

This option was of some use before HaskellDirect supported the reading of type libraries - of little use now.

`-fanon-typelib`

For Haskell COM components that implement the interface `IDispatch`, the compiler will generate a type library to aid in its implementation. This type library is normally accessed by looking up its registered GUID, but in the event you don't want to have the type library be registered, use `-fanon-typelib`.

`-fmaybe-optional-params`

For Automation clients, the compiler's default behaviour when encountering optional arguments is to use overloading. For example,

```
[...]
interface IA : IDispatch {
  HRESULT meth([in,optional]int i);
};
```

gives

```
meth :: Variant a => a0 -> IA a -> IO ()
```

the option `-fmaybe-optional-params` uses a `Maybe` instead, giving you

```
meth :: Maybe Int -> IA a -> IO ()
```

`-fignore-importlibs`

If any `importlib()` declarations are encountered in the IDL input, just ignore them.

`-fignore-methods-upto=Name`

Instead of factoring them out, some (generated) IDL input may prefix all interfaces with a set of identical methods, i.e., the interfaces contain:

```
interface IA : IUnknown { HRESULT m1 (); .... }
interface IB : IUnknown { HRESULT m1 (); .... }
```

rather than

```
interface IBase : IUnknown { HRESULT m1 (); }
interface IA : IBase { HRESULT m1 (); .... }
interface IB : IBase { HRESULT m1 (); .... }
```

To avoid generating N stubs for `m1`, use `-fignore-methods-upto=m1`.

`-fshow-idl-in-comments`

Put the IDL in comments next to the Haskell code that it generated.

`-fappend-interface-short-name`

Within each Haskell module generated, if there's a name clash between two method names, they're disambiguated by appending the number `1` to the second one. The `-fappend-interface-short-name` option takes a different approach at disambiguating methods by appending the upper case letters of their corresponding interface. For example,

```
interface IOleFoo : IUnknown {
  HRESULT meth();
};
interface IOleBar : IUnknown {
  HRESULT meth();
};
```

The stub for the method `IOleBar.meth` is named `methOB` in the generated Haskell code.

`-fno-gen-variant-instance`

For each enumeration type, the default is to generate an instance of the `Automation.Variant` class it. Use `-fno-gen-variant-instance` to turn this off.

`-fexplicit-i-pointer`

For Automation clients, the generated stubs does not give the interface pointer explicitly as an argument. To have it do so, use `-fexplicit-i-pointer`.

`-fno-library-ids`

For each `library` declaration, we generate a Haskell definition of its `LIBID`. To disable this, use `-fno-library-ids`.

`-foverload-variant`

When generating Automation client stubs, the default is to overload arguments that have type `VARIANT`. To enable this behaviour for non-Automation clients, use `-foverload-variant`.

`-fuse-std-dispatch`

When generating proxy code for Haskell COM servers, the default is to use the system-supplied, type library marshaller to implement `IDispatch`. A simpler, Haskell-only implementation of `IDispatch` is also supported by the HaskellDirect COM libraries. To use this instead, use `-fuse-std-dispatch`.

4.9 Running a pre-processor

With `-cpp` on the command-line, the IDL compiler will do a 'pre-wash' of the input by first running a pre-processor over it. Which pre-processor to use is determined as follows:

- If the `CPP` environment variable is defined, use what it is bound to as the pre-processor to invoke.
- If `CPP` isn't defined, the default value for it is `gcc -E -x c`.

- By default, the output from the pre-processor is written to a temporary file in the directory `/tmp/`. To put it elsewhere, set the environment variable `TMPDIR`.

The options `-D` and `-U` can be used to define or 'undefine' CPP variables via the command line.

To use the Microsoft Visual C++ compiler as your pre-processor, set the environment variables as follows:

```
CPP=cl /nologo /P
TMPDIR=
```

4.10 Off-line attribute specification

As an alternate mechanism to specifying the IDL attributes that are associated with a type / method, HaskellDirect has got *Attribute Specification Files (ASF)*.

Instead of specifying the attribute list in the input itself, you give the hierarchic name of the IDL name you want to configure along with the attributes you want to associate. For instance, if you feed the compiler the IDL declaration:

```
char* getenv(char*);
```

there's no telling what kind of pointer values are being passed or returned here. So, if you additionally supply the ASF information:

```
getenv : [string,nofree]
getenv.arg1 : [in,string]
```

it will be processed as if you'd written

```
[string,nofree]char* getenv([in,string]char*);
```

causing the HaskellDirect to give the `getenv()` stub the following (friendly) signature:

```
getenv :: String -> IO String
```

An ASF file is specified on the command line using the `-asf=`*`file`* option. The syntax is simply a sequence of line-separated name, attribute pairs, i.e.,

```
asf : {- empty -}   { [] }
    | asf att_pair  { $2 : $1 }

att_pair : ATTRIB_NAME ':' attributes { ($1,False,$3) }
         | ATTRIB_NAME '=' attributes { ($1,True,$3) }
```

An attribute pair can either augment the list of attributes associated with an IDL name, or replace it. `":"` does the former, while `"="` replaces. Unless you have good reasons not to, stick with `":"`.

Notice: parameters to a method/procedure can either be referenced by name or position, i.e.,

```
f.arg1 : [in]
f.sz   : [in]
```

both refer to the first argument of `int f(int sz);`.

5 Compiling the output

The output generated by HaskellDirect can be made use of in a couple of different ways:

- If HaskellDirect was used to create a Haskell binding to either an external library or some COM components, the generated output can, by default, be compiled/used with a Haskell system that supports the new FFI (plus has got a port of the support libraries that HaskellDirect makes use of). As of May 1999, ghc is the only such system, but Hugs will soon have this supported too.
- By supplying the option `-hugs`, HaskellDirect will generate code that can be used with versions of Hugs that doesn't support the Hugs / GHC FFI. This only applies to client stubs though ; if you want to write Haskell COM components (or wrap up Haskell code as a C callable library), you will need to be working with a system that supports the FFI.
- If you compiled the input with `-jni`, HaskellDirect generates code that can be used in conjunction with the Lambada Haskell-Java bridge. For instructions on how to go about using the generated stubs, please consult the Lambada documentation.
- If the specification contained custom COM interfaces, the output can be compiled with `ghc-3.03` (or `ghc-4.0x`) on Win32 platforms. `ghc-4.03` or later is recommended, but it may work with earlier releases.
- If the IDL specification contained Automation interfaces, the generated client stubs can either be compiled using ghc or used directly with *HaskellScript* `<http://www.haskell.org/active/activehaskell.html>` or Hugs.

5.1 Compiling C library stubs

For hints on what `ghc` options to use (and options to feed the IDL compiler), have a look at the `Makefiles` in the `examples` section of the distribution.

5.2 Compiling COM clients

To compile up the HaskellDirect generated COM client stubs, you'll need to supply the following command line options

```
ghc -c myStubs.hs -fglasgow-exts -i<path to hdirect libdir>
```

To build a COM client binary using ghc, you'll have to include a bunch of libraries on the command line:

```
ghc -o main main.o myStubs.o -fglasgow-exts \
-L<path to hdirect libdir> -lcom -luser32 -lole32 \
-loleaut32
```

5.3 Linking Haskell COM DLLs

To link a Haskell-implemented COM DLL server, you need to invoke GHC as follows:

```
ghc --mk-dll -o component.dll -optdll--def -optdllComServer.def <object files> \
ComDllMain.o dll_stub.o -fglasgow-exts -syslib com
```

Remarks:

- The option `-mk-dll` instructs the GHC driver to generate a Win32 DLL rather than an executable binary.
- The magic `-optdll-def -optdllComServer.def` creates a DLL which will only export the symbols mention in the file `ComServer.def`.

  This DEF file is supplied with the HaskellDirect distribution, along with the DLL wrapping code `ComDllMain.lhs` and `dll_stub.c`, which you need to compile up and supply with all Haskell COM servers.

5.4 Using Hugs

When compiling with the `-hugs` option on, a `.c` file is generated, as well as a `.hs` file. Before the Haskell stubs can be loaded by Hugs, you'll have to compile up the `.c` stubs into a dynamically loadable library.

The details of how this is done differs from platform to platform, but here's an outline of how to do it on platforms using ELF (e.g., modern versions of Linux) and Win32. Under ELF, it's just a matter of generating a little bit of position-independent code:

```
gcc -fPIC -shared -o MyModule.so MyModule.c -Ihdirect/dir
```

A couple of things to note about this:

- Use the options `-fPIC -shared` to generate a dynamically loadable library - standard stuff, really.
- The generated stub contain a `#include` of the header file `HDirect.h`, which is supplied with the HaskellDirect distribution. So, you'll need to use the `-I` option to tell the C compiler where to find it.
- It is very likely that you will need to augment the above command line with some `-l` and `-L` options to satisfy the linker ; exactly which ones depend on what library the generated stubs help you interface to.

On the Win32 side, the extension module can be created using Microsoft Visual C++

```
cl /nologo /LD /MD -o MyModule.dll MyModule.c -Ihdirect/dir
```

which also needs to be pointed at the directory containing the `HDirect.h` header file.

Alternatively on Win32 platforms, you can use `mingw32` as supplied by the Cygwin toolchain distribution, for example:

```
c:\example> gcc -mno-cygwin -o MyModule.o MyModule.c -Ihdirect/dir
c:\example> dllwrap -mno-cygwin --target=i386-mingw32 \
            --def HugsExt.def MyModule.o -o MyModule.dll
```

`HugsExt.def` is the module definition file, and contains the following two lines:

```
EXPORTS
initModule
```

This `.def` file isn't supplied, so you'll have to type it all in yourself.

As in the ELF case, both the VC++ and mingw32 route will in all likelihood require you to augment the command-lines in order to satisfy the linker.

6 HaskellDirect examples

This section shows how you can use HaskellDirect to solve some of the common Haskell interoperation problems.

6.1 Interfacing with C header files

One of HaskellDirect's front-ends support DCE IDL, which is syntactically extremely close to C prototype and type declarations in the sense that a C header file can be viewed as low-quality IDL input.

HaskellDirect supports the reading of header files together with a number of mechanisms to help you improve on the information that header file declaration conveys. To elucidate how HaskellDirect can be used, here's a set of steps you will commonly have to go through to import into Haskell the functionality specified in a header file.

- Create a wrapper IDL module which simply includes the header file, i.e.,

  ```
  module CoolAPI {
  #include "CoolAPI.h"
  };
  ```

- Compile the wrapper IDL with `ihc`, using the following options:

  ```
  ihc -fhs-to-c CoolAPI.idl -cpp -I/extra/include/path/needed
  ```

  i.e., stating that you're generating stubs that let's you call C functions from Haskell (`-fhs-to-c`), enable the use of the C pre-processor plus any configuration of the include search path that might be needed.

- If you want to use the generated stubs from Hugs, add `-hugs` to the command-line.

By following the above three steps, HaskellDirect should have generated `CoolAPI.hs`, which contains the necessary type and function stub definitions. However, more often than not, these stubs are less than satisfactory, for a number of reasons:

- Contains lot of defintions included from standard header files that you are not interested in.

  To avoid this, use the HaskellDirect option `-fexclude-system-includes`.

- The stubs aren't particularly useable from a Haskell programmer point-of-view. For example, from

  ```
  char* getenv(char*);
  ```

  HaskellDirect generates:

  ```
  getenv :: Ptr Char -> IO (Ptr Char)
  ```

  From reading the documentation of the function you *know* that the pointers in question are null-terminated character strings, but there's no way of informing HaskellDirect that this is the case, at least not via the header file. IDL does supply you with the capabilities to specify this, through its `string` attribute:

  ```
  [string]char* getenv([in,string]char *arg);
  ```

but requiring the user to convert the header file to IDL is not a practical proposition in most cases. So, to support the user here, HaskellDirect has got support for Attribute Specification Files (ASFs), which let's you specify the attributes of a function separately from its prototype/declaration. For information on how ASFs lets you improve upon the quality of the generated stub code, see Section 4.10 (Off-line attribute specification).

- The generated output doesn't include defnitions for any of the `#define`s that the header file contained.

  There are no shrink-wrapped solutions to this one (at least not yet.) Current alternatives are:

  – Make use of GNU CPP's support for dumping the #defines it encountered while processing (i.e., `-dM`), to list them all. Convert this list into corresponding IDL `const` declarations, either through a simple script or by-hand.

  – Contact HaskellDirect's author for a patch to GNU CPP that adds support for converting `#define`s to `const` declarations. With that patch added, automatic conversion of `#define`s into equivalent Haskell defintions becomes possible.

    [It is my intention to forward this patch to the GNU CPP maintainers to have it considered for inclusion, as it serves a useful role outside our (small) sphere.]

- You're only interested in a subset of the functions declared in the header file, *or* not interested in all but a couple.

  To have HaskellDirect omit stubs for certain functions, use the option `-fwinnow-defns` or `-fonly-remove-defns` together with an ASF file. For instance, assuming your header file contains prototypes for functions `f` and `g` but you're only interested in `f`. To achieve this you supply an ASF (via the option `-asf=`) containing the following on-liner:

  ```
  f:
  ```

  and compile up the stubs using:

  ```
  ihc -fhs-to-c -cpp CoolAPI.idl --asf=CoolAPI.asf -fwinnow-defns
  ```

  Since `g` isn't mentioned in the ASF, it will be ignored since `-fwinnow-defns` will only generate stub code for definitions mentioned in the ASF file. Notice that if `f` uses any locally defined types, they will also be included in the generated output.

  In the cases where it is simpler to name the defintions you want to exclude from the set of generated Haskell stubs, `-fonly-remove-defns` is your friend. For the above example, you would then supply an ASF listing the functions you want to omit:

  ```
  g=
  ```

  and feed that one-line ASF to HaskellDirect along with `-fonly-remove-defns`.

As a parting remark, it is the author's experience that it is often easy to fall into the trap of spending lots of effort in trying to supply the set of IDL attributes that makes the generated stubs look *just right* to the Haskell programmer. If you find yourself in this situation, it may be worth considering whether you should instead shift some of the responsibility of providing a Haskell-friendly veneer to the external API into Haskell itself, i.e., write the impedance matching code yourself.

6.2 Generating COM client stubs

To have HaskellDirect generate a set of stubs that allows you to invoke the methods of a COM component's interfaces from Haskell, you either need to:

- Have the component specified in IDL; some components come with the IDL included, but many do not.
- Have the type library describing the component. If you have got the type library, resist the temptation of de-compiling with an utility such as OleView; use HaskellDirect's support for reading the type library directly, since it does a better job of de-compiling the type library than OleView.

If you've got the information in IDL-form, you would then invoke HaskellDirect as follows:

```
ihc  -fcom Component.idl -o Component.hs
```

(And if you've got it in type library form, add `-tlb` to the command line.)

All being well, the above invocation should generate the file `Component.hs` containing the Haskell source code that provides the stubs to your component's methods.

If the component has got any binary interfaces (i.e., derived from `IUnknown` and not via `IDispatch`) *and* you intend to use the generated stubs with Hugs, you also need to supply the option `-hugs`.

If you're using a compiler (i.e., GHC), you then need to compile the generated Haskell source – see Section 5.2 (Compiling COM clients). If you're using Hugs, you (naturally) don't need to process the Haskell source any further, but in the event your component contained any binary interfaces, you will need to compile up the generated C stubs, see Section 5.4 (Using Hugs) for instructions how to do this.

For a worked example of how to compile up and write COM clients in Haskell, have a look at the code in the `examples/comcli` directory.

6.3 Generating Lambada stubs

Details of how to do this is covered by the Lambada user's manual.

6.4 Implementing Haskell COM servers

If you're using GHC, you can also use HaskellDirect to help you create COM components/servers implemented in Haskell. This section explains the steps you need to follow when generating a Haskell COM component.

As always, the starting point is the specification of the component's interfaces in IDL. Notice that if your component provides an `IDispatch` interface to make it possible/easy to use the component from within a scripting language such as `JavaScript`, HaskellDirect provides you with an option of how to implement the `IDispatch` wrapper. It can either generate code which uses the system-provided type-library marshaller, or use a Haskell-only solution. If you choose to go the type-library route, you will need to express the `IDispatch` interface as follows:

```
[ ... , oleautomation ]
interface IA : IUnknown { ... };

[ ..., odl, oleautomation ]
dispinterface DA { interface IA; };
```

i.e., using `interface IA : IDispatch {...};` will not work as expected.

Having specified the interfaces, the next step is to generate the proxy and the skeleton code for your component:

```
ihc -fcom MyComponent.idl -s --skeleton
```

which will generate two Haskell files, `MyComponentProxy.hs` and `MyComponent.hs`. The latter contains the skeletal implementation of the methods that your component implements, while the former provides the *proxy* code which takes care of wrapping up your methods behind a COM-callable interface.

The next step is to fill in the skeleton code with your implementation of the component. Once done, compile up `MyComponent.hs` followed by `MyComponentProxy.hs`.

With that out of the way, you now have to decide how to package your component. HaskellDirect's COM libraries provides support for building an in-proc server (i.e., in DLL form) or a local-proc server (i.e., in EXE form.)

If you go with the DLL option, you will need to copy two files from the HaskellDirect's lib directory into your project directory, `ComDllMain.lhs` and `dll_stub.c`, and compile these up.

Finally, you will need to supply a `Main` module, which configures the name of the component etc.:

```
module Main(main, comComponents) where

import MyComponentProxy
import ComDll

comComponents :: [ ComponentInfo ]
comComponents =
  [ hasTypeLib $
    withComponentName "A Component I Prepared Earlier" $
    withProgID "MyComponent.1" $
    withVerIndepProgID "MyComponent" $
    componentInfo
  ]

main = putStrLn "In Main"
```

(The `Main.main` isn't ever used, but GHC insist that there is one.]

Once the Main module has been successfully compiled, build the DLL by following the instructions provided in Section 5.3 (Linking Haskell COM DLLs)

The HaskellDirect distribution comes with a couple of examples of Haskell COM servers, `examples/comserv` shows how to build and package up an in-proc server.

If you instead want to provide a local-proc COM server in EXE form, things are a little simpler. You simply need to supply a `Main` module which starts up your server.

```
module Main(main) where

import MyComponentProxy
import ExeServer

main :: IO ()
main = do
   runExeServer MyComponentProxy.componentInfo
```

For more information on what the various libraries used in this section provides, please consult Section 7 (Support libraries).

7 Support libraries

This section contains a description of the programming interfaces provided by miscellaneous libraries that help Haskell code interact with the outside world. Libraries that help you marshal data across the boundary, write COM clients (Automation based or not), and build COM components in Haskell.

7.1 The `HDirect` library

The `HDirect` library provides basic functionality for packing and unpacking a range of basic values between their external and Haskell representations. It is used quite heavily by the generated the HaskellDirect code, but the functions it exports are often of use in other contexts.

For each of the basic types, it provides the following set of marshaling actions:

```
module HDirect where

marshallInt    :: Int -> IO Int
unmarshallInt  :: Int -> IO Int
writeInt       :: Ptr Int -> Int -> IO ()
readInt        :: Ptr Int -> IO Int
sizeofInt      :: Word32

 same for  Int{8,16,32,64}
           Word{8,16,32,64}
           Char, Addr
           Double, Float
           Bool, ForeignObj
```

The library defines five marshalling functions for each of the basic types, by-value and by-reference marshallers plus a constant holding the size of the marshalled representation of the basic type. See the HaskellDirect paper for information of how these functions are used.

To manipulate pointers in Haskell, we define a synonym, `Ptr`, to (at least) make the type signatures of the pointer manipulating functions more concisely convey their behaviour:

```
type Ptr a    = Addr  -- we could (and should) use a newtype here.

addNCastPtr   :: Ptr a -> Word32 -> Ptr b
mkPtr         :: Addr  -> Ptr a
pointerToAddr :: Ptr a -> Addr
derefPtr      :: Ptr (Ptr a) -> IO (Ptr a)
indexPtr      :: Ptr (Ptr a) -> Int -> IO (Ptr a)

allocOutPtr   :: IO (Ptr a)
allocBytes    :: Int -> IO (Ptr a)
allocWords    :: Int -> IO (Ptr a)
```

`Ptr`s are allocated using the C run-time's `malloc()`, since they need to be (immovable) pointers that we can hand over to the outside world. To free pointers, the following three actions are provided:

```
freePtr     :: Ptr a -> IO ()
trivialFree :: Ptr a -> IO ()    -- trivial freeing just
doThenFree  :: (Ptr a -> IO ())  -- freeing action
            -> (Ptr a -> IO b)   -- unmarshaller
            -> Ptr a             -- the ptr.
            -> IO b
```

with `doThenFree` capturing the pattern of unmarshalling a pointer value followed by the release of it. `trivialFree` is the nullary freeing action, it doesn't call `free()` on its argument, just ignores it.

DCE IDL categorises pointers into three kinds, `ptr`s, `unique`s and `ref`s – see IDL documentation and the *HDirect paper* `<http://www.dcs.gla.ac.uk/~sof/hdirect.ps.gz>` for more information of what distinguishes them. The `HDirect` library provides the implementation of the pointer marshalling schemes given in that paper – here's the `ptr` marshallers:

```
marshallptr   :: Ptr a -> IO (Ptr a)
unmarshallptr :: Ptr a -> IO (Ptr a)
writeptr      :: Ptr (Ptr a) -> Ptr a -> IO ()
readptr       :: Ptr (Ptr a) -> IO (Ptr a)

writefptr     :: Ptr ForeignObj -> ForeignObj -> IO ()
```

A unique pointer is represented in Haskell as a `Maybe` value:

```
marshallunique :: IO (Ptr a)              -- allocate
               -> (Ptr a -> a -> IO ())   -- marshall into
               -> Maybe a                 -- value
               -> IO (Ptr a)
unmarshallunique :: (Ptr a -> IO a)       -- unmarshall value
                 -> Ptr a                 -- unique ptr
                 -> IO (Maybe a)
writeunique   :: IO (Ptr a)               -- allocate
              -> (Ptr a -> a -> IO ())    -- marshall into
              -> Ptr a                    -- (ptr to) unique ptr
              -> Maybe a
              -> IO ()
readunique    :: (Ptr a -> IO a)   -- unmarshall value
              -> Ptr (Ptr a)       -- unique ptr.
              -> IO (Maybe a)
freeunique    :: (Ptr a -> IO ())
              -> Ptr (Ptr a)
              -> IO ()
```

Reference (`ref`) pointers are invisible in Haskell, representing them by the value they point to:

```
marshallref  :: IO (Ptr a)
             -> (Ptr a -> a -> IO ())
             -> a
             -> IO (Ptr a)
unmarshallref :: (Ptr a -> IO b)
              -> Ptr a
              -> IO b
writeref      :: IO (Ptr a)
              -> (Ptr a -> a -> IO ())
```

```
                -> Ptr (Ptr a)
                -> a
                -> IO ()
readref         :: (Ptr a -> IO a)
                -> Ptr (Ptr a)
                -> IO a
freeref         :: (Ptr a -> IO ())
                -> Ptr b
                -> IO ()
```

The generated HaskellDirect source maps IDL enumerations onto Haskell enumeration types (and define `Enum` instances over them). To marshall such values across the boundary, the following helper functions are used:

```
marshallEnum32   :: Enum a => a -> IO Int32
unmarshallEnum32 :: Enum a => Int32 -> IO a
writeEnum32      :: Enum a => Ptr Int32 -> a -> IO ()
readEnum32       :: Enum a => Ptr Int32 -> IO a

marshallEnum16   :: Enum a => a -> IO Int16
unmarshallEnum16 :: Enum a => Int16 -> IO a
writeEnum16      :: Enum a => Ptr Int16 -> a -> IO ()
readEnum16       :: Enum a => Ptr Int16 -> IO a
```

To marshall lists of values, the following set of actions are provided:

```
marshalllist :: Word32                 -- size of a (marshalled) value
             -> (Ptr a -> a -> IO ())  -- (reference) marshaller for a value
             -> [a]                    -- values
             -> IO (Ptr [a])

unmarshalllist :: Word32               -- size of a (marshalled) value
               -> Word32               -- starting offset (in sizeof units).
               -> Word32               -- number of values to unmarshal
               -> (Ptr a -> IO a)      -- (reference) unmarshall for a value
               -> Ptr [a]
               -> IO [a]

writelist      :: Bool                  -- do we have to allocate?
               -> Word32                -- size of a (marshalled) value
               -> (Ptr a -> a -> IO ()) -- (reference) marshaller.
               -> Ptr [a]               -- where to put the marshalled list.
               -> [a]
               -> IO ()

readlist       :: Word32                -- size of a (marshalled) value.
               -> Word32                -- number of elements to unmarshal
               -> (Ptr a -> IO a)       -- (reference) unmarshaller.
               -> Ptr [a]
               -> IO [a]

freelist       :: Word32                -- size of a (marshalled) value.
               -> Word32                -- number of elements
               -> (Ptr a -> IO ())      -- how to free an element.
```

```
                 -> Ptr [a]
                 -> IO ()
```

We provide a special set of marshallers for `String`, which map to and from the external representation of a zero-terminated sequence of (8-bit byte) characters:

```
marshallString    :: String -> IO (Ptr String)
unmarshallString  :: Ptr String -> IO String
readString        :: Ptr (Ptr String) -> IO String
writeString       :: Bool -> Ptr String -> String -> IO ()
freeString        :: Ptr String -> IO ()
```

A special case of the string marshallers allow you to express the number of characters you want to marshall across (in either direction):

```
marshallBString   :: Int  -> String -> IO (Ptr String)
unmarshallBString :: Int  -> Ptr String -> IO String
writeBString      :: Bool -> Int -> Ptr String -> String -> IO ()
readBString       :: Int  -> Ptr (Ptr String) -> IO String
```

Restricting the marshalling to a fixed number of elements in a list is sometimes useful for other list values too, so we provide the following set:

```
marshallblist :: Word32
              -> Word32
              -> (Ptr a -> a -> IO ())
              -> [a]
              -> IO (Ptr [a])
writeblist    :: Word32
              -> Word32
              -> (Ptr a -> a -> IO ())
              -> Ptr [a]
              -> [a]
              -> IO ()
readblist     :: Word32
              -> Word32
              -> (Ptr a -> IO a)
              -> Ptr a
              -> IO [a]
```

7.2 The `WideString` library

With the notable exception of HBC, Haskell systems represent characters internally via 8-bit bytes and not with wider formats capable of holding Unicode characters in raw form (16-bit or more). To help users on these imparied systems interface with the external world that expect "wide strings", the `WideString` library can be used. It provides the following:

```
newtype WideString -- abstract

nullWideString       :: WideString
stringToWide         :: String     -> IO WideString
wideToStr            :: WideString -> IO String
```

```
wideStringLen        :: String     -> Word32
word16ToWideString   :: Word16 -> IO WideString

lengthWideString     :: WideString -> Int

marshallWideString   :: WideString -> IO Addr
unmarshallWideString :: Ptr WideString -> IO WideString
writeWideString      :: Ptr WideString -> WideString -> IO ()
readWideString       :: Ptr WideString -> IO WideString
freeWideString       :: Ptr WideString -> IO ()
sizeofWideString     :: Word32
```

Notice that `WideString`s are not currently finalised for you by the garbage collector, so you have to release them explicitly.

7.3 The Com library

The Com library offers support for performing various basic COM programming tasks, akin to what the Com library that's present on all Win32 boxes offers to C/C++ programmers. The types and functions presented in this section are all exported by the Haskell module `Com`.

7.3.1 Com exceptions

In COM, methods communicate status back to their caller by returning an `HRESULT`, a 32-bit value indicating rudimentary information about success or failure of a call. The Haskell Com library allows you to construct `HRESULT`s and provides an interface between Haskell's IO exceptions and `HRESULT`s. The programming interface provided is:

```
type HRESULT = Int32

checkHR    :: HRESULT  -> IO ()
   -- failure => raise IO exception.
returnHR   :: IO ()    -> IO HRESULT
   -- exception raised => return E_FAIL.
   -- if not, return S_OK.

coFail     :: HRESULT -> IO a
coFailWith :: HRESULT -> String -> IO a
   -- Raise (HRESULT) IO exceptions.

isCoError    :: IOError -> Bool
getCoErrorHR :: IOError -> Maybe HRESULT

coGetErrorString :: IOError -> String

winErrorToHR :: Int32 -> HRESULT
getLastError :: IO Word32

succeeded :: HRESULT -> Bool
failed    :: HRESULT -> Bool

hresultToString :: HRESULT -> IO String
```

On top of that, the module `ComException` defines all the HRESULT exception codes that the COM and OLE APIs use.

7.3.2 The `GUID` type

So-called globally unique identifiers are used throughout by COM, and the following functions are provided for manipulating them in Haskell:

```
data GUID = ... -- abstract
  -- instance of Eq and Show

mkGUID        :: String -> GUID
guidToString  :: GUID   -> String
stringToGUID  :: String -> IO GUID
nullGUID      :: GUID

marshallGUID   :: GUID -> IO ForeignObj
unmarshallGUID :: Bool -> Ptr GUID -> IO GUID
copyGUID       :: GUID -> IO Addr
writeGUID      :: Ptr GUID -> GUID -> IO ()
readGUID       :: Bool -> Ptr GUID -> IO GUID
sizeofGUID     :: Word32
```

7.3.3 The `CLSID` type

One use of `GUID`s are as unique identifiers of COM components / classes, so called `CLSID`s. To help catch any accidental use of a GUID that doesn't represent a `CLSID`, we make `CLSID` a separate Haskell type from a `GUID`:

```
data CLSID = ... -- abstract
  -- instance of Eq and Show

mkCLSID            :: String -> CLSID
stringToCLSID      :: String -> IO CLSID

clsidToDisplayName :: CLSID  -> String

guidToCLSID        :: GUID   -> CLSID
clsidToGUID        :: CLSID  -> GUID

marshallCLSID      :: CLSID  -> IO ForeignObj
copyCLSID          :: CLSID -> IO Addr
unmarshallCLSID    :: Bool   -> Ptr CLSID -> IO CLSID
writeCLSID         :: Ptr CLSID -> CLSID -> IO ()
readCLSID          :: Bool   -> Ptr CLSID -> IO CLSID
sizeofCLSID        :: Word32
```

The set of operations are largely identical to what is provided for `GUID`s, but with the addition of `clsidToDisplayName`, which converts a `CLSID` into the display string expected by the standard class moniker.

7.3.4 The IID type

To identify COM interfaces, the IID type is used; yet another use of GUIDs. The representation of IIDs in Haskell is interesting:

```
data IID ip = ... -- abstract
  -- instance of Eq and Show

mkIID       :: String -> IID a
stringToIID :: String -> IO (IID a)
guidToIID   :: GUID   -> IID a
iidToGUID   :: IID a  -> GUID

marshallIID      :: IID a -> IO ForeignObj
copyIID          :: IID a -> IO Addr
unmarshallIID    :: Bool   -> Ptr (IID a) -> IO (IID a)
writeIID         :: Ptr (IID a) -> IID a -> IO ()
readIID          :: Bool   -> Ptr (IID a) -> IO (IID a)
sizeofIID        :: Word32
```

An IID is parameterised over the type of its corresponding interface pointer (see next section for how we represent them in Haskell). By parameterising the unique identifier over its type, we're able to ensure that there's no mismatch between asking for a pointer to a particular interface and the subsequent use of that pointer, i.e., method application is type safe - a nice property to have.

7.3.5 Interface pointers

A COM interface pointer is represented by the Haskell type IUnknown:

```
data IUnknown_ iface = .. -- abstract
type IUnknown  iface = IUnknown_ iface

ifaceToAddr    :: IUnknown a -> Addr
addrToIPointer :: Bool       -> Addr -> IUnknown a
          -- True => attach finaliser that calls Release().

marshallIUnknown   :: IUnknown a -> IO ForeignObj
unmarshallIUnknown :: Bool -> Addr -> IO (IUnknown a)
    -- True => call AddRef() on the i-pointer.
readIUnknown       :: Bool -> Addr -> IO (IUnknown a)
    -- True => call AddRef() on the i-pointer.
writeIUnknown      :: Bool -> Addr -> IUnknown a -> IO ()
    -- True => call AddRef() on the i-pointer.
```

The interface pointer is parameterised over the particular interface it is a pointer to. To illustrate, assuming you have got the following two IDL interface declarations:

```
[...]interface IA : IUnknown { HRESULT m1(); }
[...]interface IB : IA       { HRESULT m2(); }
```

When fed this as input, HaskellDirect will generate the following definitions for the two interface pointers:

```
data IA_ a = IA_
type IA  a = IUnknown (IA_ a)

data IB_ a = IB_
type IB  a = IA (IB_ a)
```

The `IA` type synonym encodes that `IA` extends the `IUnknown` interface. The type argument it takes encodes any possible extensions of `IA`, as witnessed on the right hand side of the definition of `IB`.

To state that an interface pointer is of a particular interface, we use the unit type as a 'plug':

```
iidIUnknown :: IID (IUnknown ())
```

The `iidIUnknown` represent the IID for just the `IUnknown` interface; no more, no less.

The extra type parameter is particularly useful when generating method stubs. For example, the above two interface declarations will cause HaskellDirect to generate the following stubs:

```
m1 :: IA a -> IO ()
m2 :: IB a -> IO ()
```

A separate stub for invoking method `m1` on an `IB` interface pointer isn't required, you can safely re-use the one generated for `IA`.

7.3.6 The `IUnknown` interface

The `Com` library provides wrappers for the methods of the base COM interface, `IUnknown`:

```
queryInterface :: IID (IUnknown b)   -- IID of interface you want to get at.
               -> IUnknown a         -- interface pointer you've got
               -> IO (IUnknown b)    -- the desired interface pointer.

addRef         :: IUnknown a -> IO Word32
release        :: IUnknown a -> IO Word32

iidIUnknown    :: IID (IUnknown ())

interfaceNULL  :: IUnknown a
iidNULL        :: IID ()
```

The methods that handle the reference counting should not really be used, since interface pointers are *finalized* by the garbage collector, meaning that when an interface pointer is no longer used by a Haskell program, the garbage collector will call `release` on the object for you.

The fundamental method of COM is `queryInterface`, and notice how the type parameters match up between the `IID` and the interface pointer that's returned, ensuring the type safety of this method.

7.3.7 The `BSTR` type

Many COM servers and clients use a length-prefixed representation of strings, via the `BSTR` type. Support the conversion between Haskell strings and `BSTR` via:

```
marshallBSTR   :: String -> IO (Ptr BSTR)
unmarshallBSTR :: Ptr BSTR -> IO String
writeBSTR      :: Ptr BSTR -> String -> IO ()
readBSTR       :: Ptr BSTR -> IO String
freeBSTR       :: Ptr BSTR -> IO ()
```

7.3.8 Creating component instances

To create objects, the `Com` library mirror what the standard COM library provides:

```
coCreateInstance :: CLSID
                 -> Maybe (IUnknown b)
                 -> CLSCTX
                 -> IID (IUnknown a)
                 -> IO (IUnknown a)
```

See COM documentation for the complete story on `CoCreateInstance()`, but here's a synopsis: First argument is the `GUID` of the component you wish to create, the second is the interface pointer to the *aggregating* component. Its use is not relevant in the vast majority of cases, so just pass a `Nothing` here. The third argument is of type `CLSTX`, which lets you specify how you want the component created:

```
data CLSCTX
 = CLSCTX_INPROC_SERVER
 | CLSCTX_INPROC_HANDLER
 | CLSCTX_LOCAL_SERVER
 | CLSCTX_INPROC_SERVER16
 | CLSCTX_REMOTE_SERVER
 | CLSCTX_INPROC_HANDLER16
 | CLSCTX_INPROC_SERVERX86
 | CLSCTX_INPROC_HANDLERX86
 | LocalProcess
 | InProcess
 | ServerProcess
 | AnyProcess
```

A component can be instantiated inside your process (`InProcess`), or in a separate one (`ServerProcess`) plus a couple of nuances inbetween. If you don't care where, use `AnyProcess` and `coCreateInstance` will pick the most appropriate one for you.

The fourth argument to `coCreateInstance` is the IID of the interface you want to initially create the component at.

If the component has been installed and registered on your machine and it can be instantiated, `coCreateInstance` returns an interface pointer upon (successful) return. To summarise, here's a simple example of its use:

```
main = do
    ip <- coCreateInstance
            (mkCLSID "{99B42120-6EC7-11CF-A6C7-00AA00A47DD2}")
            Nothing
            AnyProcess
            (mkIID "{973F77A0-6EC7-11CF-A6C7-00AA00A47DD2}")
    putStrLn "Created component"
    return ()
```

In addition to `coCreateInstance`, the Com library also supplies a couple of other component creating actions, which are at times more convenient to use than `coCreateInstance`:

```
coCreateObject   :: ProgID
                 -> IID (IUnknown a)
                 -> IO (IUnknown a)

coGetFileObject  :: String
                 -> ProgID
                 -> IID (IUnknown a)
                 -> IO (IUnknown a)

coGetActiveObject :: ProgID
                  -> IID (IUnknown a)
                  -> IO (IUnknown a)

coGetObject      :: String
                 -> IID (IUnknown a)
                 -> IO (IUnknown a)

type ProgID = String

progIDFromCLSID  :: CLSID  -> IO ProgID
clsidFromProgID  :: ProgID -> IO CLSID
```

7.3.9 Utilities

The `Com` library also provide actions for accessing the COM task allocator,

```
coAlloc :: Int  -> IO Addr
coFree  :: Addr -> IO ()
```

which you should use if you exchange (allocated) data with other components.

To emulate OO-style syntax, the `Com` library defines a pair of operators:

```
( # )   :: a -> (a -> IO b) -> IO b
( ## )  :: IO a -> (a -> IO b) -> IO b

infixl 1 #
infixl 0 ##
```

allowing you to invoke methods as follows:

```
createNew :: IA a -> IO String
createNew ip = do
  x <- ip # m1 "new"
  y <- x  # m2 1 (2.0::Double)
  return y
```

Most importantly when accessing COM objects is to remember to initialise COM itself before you do so (and ‘uninitialise’ it when you’re done). Simplest way of doing this in Haskell is to use `coRun`:

```
coRun          :: IO a -> IO ()
coInitialize   :: IO ()
coUnInitialize :: IO ()
```

You simply wrap this around your Main.main action,

```
main :: IO ()
main = coRun $ do
   ...as before...
```

and it takes care of calling `coInitialize` at the start and `coUnInitialize` at the end.

7.4 The `Automation` library

To make it easier for scripting languages to access and utilise COM components, a separate 'technology' exists called Automation built on top of the basic COM component model.

Automation eases the task of using components from within a scripting language in that method invocation is done by name, rather than by performing indexed jumps via an interface pointer.[3] Being able to invoke methods by name has more of a dynamic flavour to it, allowing for later binding.

See Automation documentation for other arguments as to why it is a Good Thing. To us Haskellers, Automation is a Good Thing in the sense that there's scores of Automation objects 'out there' that we can immediately interface to if we can 'do Automation'. Enter the `Automation` library - it provides a comprehensive and Haskell friendly view of Automation (and the interface that implements it, `IDispatch`).

7.4.1 The `IDispatch` interface

At the heart of Automation is the `IDispatch` interface, which collectively lets you invoke methods by name. The IID and types used for `IDispatch` are as follows:

```
data IDispatch_ a = IDispatch_
type IDispatch a  = IUnknown (IDispatch_ a)

iidIDispatch :: IID (IDispatch ())
```

Actually, when push comes to shove, Automation and `IDispatch` doesn't perform method invocation by name, but via (unique) numbers, so-called *dispatch identifiers* (`DISPID`s). The `IDispatch` interface provides a separate method for mapping between method names and their `DISPID`s:

```
type Member = String
type DISPID = Word32

getMemberID :: Member -> IDispatch a -> IO DISPID
```

The `DISPID` for the methods of a published interface is considered constant, so you can, if you wish, perform method invocation via those instead - the `Automation` library gives you the alternative.

[3] There's no intrinsic reason why a scripting language couldn't also do this, it's just harder and requires extra effort by the scripting client (at least that's the claim!)

7.4.2 The `Variant` class

Central to `IDispatch` is the `VARIANT` type, which is a big and ugly union/sum type that encodes the range of possible types an Automation object can be passed as arguments and return as results. In Haskell, we do not provide a type which mirrors `VARIANT` directly, instead a type class is put to good use:

```
class Variant a where
  inVariant      :: ArgIn a
  inVarList      :: ArgIn [a]
  inVarIUnknown  :: ArgIn (IUnknown a)
  vtEltType      :: a -> VARENUM
  resVariant     :: ArgRes a
  defaultVariant :: a

  resVarList      :: ArgRes [a]
  resVarIUnknown  :: ArgRes (IUnknown a)
  resVarIDispatch :: ArgRes (IDispatch a)
```

What the individual class methods do is only of interest should you want to declare your own instance - something you shouldn't have to do :), since the `Automation` library supplies the instances for you. But, unsurprisingly, what the class methods collectively provide is the bi-directional conversion between a Haskell value and the `VARIANT` values that an Automation object expects to see, hiding the details of that type from direct view to the Haskell programmer.

The important thing to notice is that if you have an overloaded type signature of the form

```
addVals :: (Variant a, Variant b)
        => a
        -> b
        -> IDispatch i
        -> IO ()
addVals x y = ...
```

and you invoke the method giving it arguments of one of the following types,

```
Char
String
IUnknown a
IDispatch a
Bool
Int
Int32
Float
Double
```

the type checker will successfully resolve the overloading and supply the dictionary containing the methods which marshals the arguments of the applied type into `VARIANT` form.

7.4.3 Invoking Automation methods

The right hand side of the `addVals` stub in the previous section is normally generated by HaskellDirect for you, but writing it out by hand is not much work, and useful should you only want to use a selection of the methods that a component exposes. Here's how to do it - suppose the `addVals` method was specified in IDL as follows:

```
[id(1)]HRESULT addVals([in]int i, [in]int j);
```

The `Automation` library supplies a family of actions for invoking Automation methods, grouped by the number of results that the methods return. There's no results here, i.e., `[out]` parameters, so we make use of `method0`:

```
method0 :: String   -- method name
        -> [VarIn]
        -> IDispatch i
        -> IO ()

addVals :: Int -> Int -> IDispatch i -> IO ()
addVals i j = method0 "addVals" [inInt i, inInt j]
```

Input parameters are passed via a list of `VarIn` values. The `VarIn` type is itself abstract, but you construct values of it using a family of coercion functions, all of the form:

```
inTy ::  Ty -> VarIn
```

where *Ty* is the name of one of the types that are instance of the `Variant` class. If you prefer to instead overload the arguments to `addVals`, you do that by using `inVariant`:

```
addVals :: (Variant a, Variant b)
        => a
        -> b
        -> IDispatch i
        -> IO ()
addVals i j = method0 "addVals" [inVariant i, inVariant j]
```

It's worth remarking here that one important feature of Automation is that it is by-design accommodating to typeless programming environments (not surprising, considering the VB development team at Microsoft implemented it in the first place!). So, as long as you can coerce your arguments into a `VARIANT`, you can pass it to an Automation component, *even if* the IDL / type information for the method states that it expects an `int` (say). The burden of converting the incoming `VARIANT` arguments into an `int` is on the Automation object/server, not its client.

Automation methods can either be accessed by name (as `method0` did), or via a unique (per interface) dispatch identifier, `DISPID`. If you want to do the latter, and save having to look it up each time you invoke the method, use the `methodID`*n* family of functions instead:

```
addVals :: Int -> Int -> IDispatch i -> IO ()
addVals i j = methodID0 1 [inInt i, inInt j]
```

It's part of the contract of an interface not to change the DISPID associated with its methods once it has been published.

There's one further twist to method invocation - if the last argument is an `[out,retval]`, you'll need to use `function`*n* instead,

```
[id(2)]HRESULT addVals([in]int i,[in]int j,[out,retval]int* res);
```

This method is invoked using `function1`:

```
function1 :: Member
          -> [VarIn]
          -> ArgOut a1
          -> IDispatch i
          -> IO a1

addVals :: Int -> Int -> IDispatch i -> IO Int
addVals i j = function1 "addVals" [inVariant i, inVariant j] outInt
```

7.4.4 Accessing properties

Automation objects has special support for setting and getting properties on an object. The `Automation` library allows you to access properties through `propertyGet` and `propertySet`:

```
propertySet :: Member -> [VarIn] -> IDispatch d -> IO ()
propertyGet :: Member -> [VarIn] -> ArgOut a -> IDispatch d -> IO a
```

For example, the property `value`,

```
[..]
interface IValue : IDispatch {
[propget,id(3)]HRESULT value([out]int* i);
[propset,id(3)]HRESULT value([in]int i);
 ...
};
```

can be accessed by the following pair:

```
getValue :: IValue a -> IO Int
getValue = propertyGet "value" [] outInt

setValue :: Int -> IValue a -> IO ()
setValue i = propertySet "value" [inInt i]
```

As with methods, properties can also be accessed via their DISPIDs:

```
propertyGetID :: DISPID -> [VarIn] -> ArgOut a -> IDispatch d -> IO a
propertySetID :: DISPID -> [VarIn] -> IDispatch d -> IO ()
```

7.4.5 Automation utilities

The library provides a couple of Automation-friendly versions of the component creation actions that the `Com` library supplies:

```
createObject    :: ProgID -> IO (IDispatch a)
getFileObject   :: String -> ProgID -> IO (IAutomation a)
getActiveObject :: ProgID -> IO (IAutomation a)
getObject       :: String -> IO (IAutomation a)

type ProgID = String
```

where a `ProgID` is a semi-readable string with the following format `<Vendor>.<Component>.<Version>` (if you don't have supply the version number, you're implicitly referring to the latest version of a component installed on your machine). An example of a `ProgID` would be `MSWinsock.Winsock`, and offer a somewhat legible way of referring to components in your programs.

7.5 The `ComServ` library

This section presents `ComServ`, a Haskell library for creating COM components. The library is intended to be used by the code generated by a tool such as HaskellDirect and applications that want to directly and dynamically create COM component instances.

7.5.1 Some COM Concepts

A COM component supports one or more *interfaces*. COM defines the binary representation of a pointer to (an instance of) an interface as follows:

```
i-pointer     +-----+        method table
----------->  |     +-------->+-------+
              +-----+         |       +-----> QueryInterface()
                              +-------+
                              |       +-----> AddRef()
                              +-------+
                              |       +-----> Release()
                              +-------+
                              |       |
                              +-------+
                              |       |
                              +-------+
```

A pointer to an interface is represented as a pointer to a pointer to a method table (vtable, for short.) How the state of the object/ component instance is represented isn't prescribed by COM, but a common (and sensible :-) strategy is to store it next to the pointer the method table.

The state of a component instance is shared by all the interfaces that the component supports.

To navigate between the different interfaces a component supports, COM guarantees that all components provides an implementation of the interface `IUnknown`. Through it, you can get from one interface to any other (that the component supports.)

A COM interface can extend/inherit another; indeed, all COM interfaces extend the `IUnknown` interface. At the level of method tables, interface extension is simply method table extension. Consequently, all COM method tables will have in their first three entries the implementation of the `IUnknown` methods.

7.5.2 Building a method table

Building an interface pointer is a two staged affair, the first of which is the construction of the table of methods that a particular interface supports. A method table is represented by the Haskell type:

```
data VTable iid objState = -- abstract, not finalised.
```

The method table has to be allocated in a chunk of memory that's not moveable; i.e., it's likely to be allocated somewhere external to the Haskell heap. So, while abstract, the concrete representation of the method table is likely to be a pointer to this piece of memory.

To make the manipulation of such pointers a tad safer, we parameterise them with respect to the type of the interface that a method table is an implementation of, and the state of the component which supports that interface.

Method tables are constructed using the following library provided action:

```
createVTable :: [Addr] -> IO (VTable iid st)
```

It creates a table big enough to hold all the method pointers of its argument list, filling in the table from top to bottom.

When creating method tables representing COM interfaces, the top three entries of all method tables collectively implement the `IUnknown` interface. You normally won't be interested in implementing these methods, so the library provides a default implementation of them.

To create a method table with the standard `IUnknown` methods prepended, the library provides:

```
createComVTable :: [Addr] -> IO (ComVTable iid objState)

type ComVTable iid objState = VTable iid (objState,IUnkState)
```

It returns a `ComVTable`, which is a standard method table but with an object state that also includes the state required to implement `IUnknown`. `IUnkState` is an abstract type.

Remarks:

- Method table sharing is desirable. Sharing across instances of a component, definitely. Sharing across implementations of an interface, possibly.

  A consequence of this is that the methods in a vtable will have to be stateless, that is, they cannot be partially applied to the object state they work over, but have to fetch that via the `this` pointer instead.

- A `VTable` could easily maintain the number of entries it has. With that in hand, we could then provide basic method table operators (e.g., union, that's either `IUnknown` savvy or not.)

- Method tables aren't currently finalised. There are no major reasons for this, other than it simplifies the standard implementation of `IUnknown`.

- The method table constructors don't have to be `IO` actions, they could be functions. However, since they're commonly used from within an `IO` context, we keep them as actions (for now.)

7.5.3 Building a component

With a method table in hand, we can now create component instances/objects:

```
createInstance :: objState
               -> VTable iid objState
               -> IO (IUnknown iid)
```

Given the state of the component instance and its method table, this action does the bare minimum of creating an interface pointer. The interface pointer has type `IUnknown iid`, parameterising the pointer type over the interface it implements.

Using `createInstance`, you can then provide your own Haskell implementation of any COM interface (*including* `IUnknown`.) Regarding `IUnknown`; to let your implementation of `QueryInterface()` be able to create new interface pointers from old, the following operation is also provided

```
cloneIPointer :: VTable iid_new objState
              -> IP iid_old
              -> IO (IP iid_new)
```

However, most of the time you'll be more than happy (we hope ;-) with using the library provided implementation of `IUnknown`, in which case you'll be wanting to use the `createComInstance` constructor instead:

```
createComInstance :: String
                  -> objState
                  -> IO ()
                  -> [ComInterface objState]
                  -> IID (IUnknown iid)
                  -> IO (IUnknown iid)
```

The arguments to `createComInstance` are:

- The file path to the Win32 DLL containing the component.
- The initial state of the component instance.
- An action to perform when the component is eventually released.
- A list of interface that the component supports.
- The interface id of the interface to initially create the component instance at.

Since all the interfaces that a component supports work over the same object state, we parameterise them over that very state using the `ComInterface` type:

```
data ComInterface objState = -- abstract
mkIface :: IID iid
        -> ComVTable iid objState
        -> ComInterface objState
```

You associate a particular interface ID with the method table that implements it by applying `mkIface`. The method table was constructed using `createComVTable`.

Back to `createComInstance`, it also takes the interface ID of the interface we want to initially create the component at. If that interface cannot be found in the list of interfaces that the component supports, an IO exception is raised.

Remarks:

- An interface pointer is parameterised with respect to its interface, but not the state of the component.
- Should `mkIface` work over `VTables` or `ComVTables`?
- No finaliser/destructor arguments to any of the component instance constructors any longer. This can be provided separately using the new Weak pointer interface, attaching a weak pointer to the object state.

7.5.4 Accessing the object state

To get at their object state, methods use:

```
type PrimIP a = Addr

getComObjState  :: PrimIP (objState, IUnkState) -> IO objState
getObjState     :: PrimIP objState -> IO objState
```

What's the difference between the two? `getComObjState` is used by methods contained in vtables that have been created via `createComVTable`, whereas `getRealObjState` is used by methods of DIY components that had their method tables constructed with `createVTable`.

Safe? I think not, any `Addr` could be used here. However, since the vast majority of the code that calls either of these functions will be automatically generated, I can live with this.

Remarks:

- The operations for getting at the state of a component instance is done by indexing off of a *primitive* interface pointer and not an `IUnknown`. Why? Because, effectively, a `PrimIP` is the 'this' pointer to a COM method, so we might as well perform the unsafe operation of fetching the component state immediately, rather than waste our time dressing it first up as an `IUnknown` before performing the unsafe op. Fetching the component state is the only operation that a method proxy uses the the 'this' pointer for.
- A `PrimIP` can be converted into an interface pointer, `IUnknown`, through the use of `Com.addrToIface`.

7.5.5 Creating dispatch interfaces

The `ComServ` library also supports the creation of components that implement the scripting-friendly `IDispatch` interface. You can create both 'pure' and 'dual' dispinterfaces:

```
createDispInterface :: IUnknown iid
                    -> Either LIBID String
                    -> IID iid
                    -> IO (IDispatch ())

createDualInterface :: StablePtr objState
                    -> ComVTable iid objState
                    -> Either LIBID String
                    -> IID iid
                    -> IO (IDispatch ())
```

The arguments to `createDispInterface` are:

- An pointer to the interface which implements the methods you want to expose via `IDispatch`.
- Where to find the *type library* which is used to drive the implementation of `IDispatch`.
- The IID of the interface which the first is an implementation of.

The creation of dual interface is a little bit more complex, since we have to combine together the `IDispatch` method table with the table that actually implements the functionality. The vtable argument that `createDualInterface` implements is assumed to be `IUnknown` derived.

The details of how to call `createDispInterface` and `createDualInterface` is taken care of by the implementation of `IUnknown` that `ComServ` provides. To indicate that a particular interface implementation is of a `IDispatch` nature, use the following two constructors

```
mkDispIface, mkDualIface :: Maybe LIBID
                         -> IID iid
                         -> VTable iid objState
                         -> ComInterface objState
```

instead of `mkIface`.

7.5.6 An example

To demonstrate how to use the above services, here's an example interface we want to provide an implementation of in Haskell:

```
interface IWho : IUnknown {
   HRESULT SongTitle ( void );
};
```

The component supports just the one interface, `IWho`, containing the single method `SongTitle`. It's implemented as follows:

```
module Who where

import MessageBox

songTitle :: String -> IO ()
songTitle title = messageBox title
```

Next step is to create the method table for the component. To do this we make use of the Hugs/GHC foreign function interface:

```
iWho_vtbl :: (ComVTable (IWho a) String)
iWho_vtbl = unsafePerformIO $ do
   s_title <- mkSongTitle (primSongTitle)
   createComVTable [s_title]

primSongTitle :: PrimIP String -> IO HRESULT
primSongTitle this = do
  title <- getComObjState this
  songTitle title
  return s_OK

foreign export stdcall dynamic mkSongTitle :: (Addr -> IO HRESULT) -> IO Addr
```

Creating an instance of the component is straightforward too:

```
new ::  String -> IO (IUnknown (IWho ()))
new initState =
  createComInstance ""
                    initState
                    (return ())
                    [mkIface iidIWho iWho_vtbl]
                    iidIWho
```

To use this component from Haskell, let's assume that we've used HaskellDirect to generate a client stub for `songTitle`:

```
module Main(main) where

import qualified Who ( new )

-- HaskellDirect generated *client* stub.
```

```
songTitle :: IWho a -> IO ()
songTitle = ...

main :: IO ()
main = do
  ip1 <- Who.new "My Generation"
  ip2 <- Who.new "Much Too Much"
  ip1 # songTitle
  ip2 # songTitle
```

A very indirect way of calling a Haskell function :-)

7.5.7 API signature

To summarise, here's the module signature for `ComServ`:

```
module ComServ where

import Com ( IUnknown, PrimIP, IID )

data ComInterface objState = --abstract
mkIface     :: IID iid
            -> ComVTable iid objState
            -> ComInterface objState

mkDispIface :: Maybe LIBID
            -> IID iid
            -> ComVTable iid objState
            -> ComInterface objState

mkDualIface :: Maybe LIBID
            -> IID iid
            -> ComVTable iid objState
            -> ComInterface objState

data IID iid -- defined by Com library.
data VTable iid Com = .. -- externally allocated

type ComVTable iid objState = VTable iid (objState, IUnkState)

data IUnkState = ... -- abstract

type PrimIP objState = Addr
  -- raw interface pointer (i.e., *not* finalised.)
  -- Com library should also define this.

createInstance :: objState
               -> VTable iid objState
               -> IO (IUnknown iid)

createComInstance :: String
                  -> objState
                  -> IO ()
                  -> [ComInterface objState]
```

```
                  -> IID (IUnknown iid)
                  -> IO (IUnknown iid)


createVTable    :: [Addr] -> IO (VTable iid objState)
    -- prepends 'standard' IU implementation.
createComVTable :: [Addr] -> IO (ComVTable iid objState)

cloneIPointer   :: IUnknown a -> VTable b -> IO (PrimIP b)

getObjState      :: PrimIP (objState, IUnkState) -> IO objState
getRealObjState  :: PrimIP objState -> IO objState

createDualInterface :: StablePtr objState
                    -> ComVTable (IUnknown iid) objState
                    -> Either LIBID String
                    -> IID (IUnknown iid)
                    -> IO (IUnknown iid)

createDispInterface :: IUnknown iid
                    -> Either LIBID String
                    -> IID (IUnknown iid)
                    -> IO (IUnknown iid)
```

7.6 The StdDispatch library

The ComServ provides one way of creating components supporting the IDispatch interface, using a type library and the COM library provided type library marshaller to do the bulk of the work.

The StdDispatch library offers a lighter-weight alternative, providing a Haskell implementation of IDispatch. The methods that your IDispatch interface supports is specified directly in Haskell, no need for type libraries. You create an instance using one of the following actions:

```
createStdDispatch :: objState
                  -> [DispMethod objState]
                  -> IID iid
                  -> IO (IUnknown iid)

data MethodKind
  = Method
  | PropertyGet
  | PropertyPut

type DISPIDLookupFun = String -> Maybe DISPID

createStdDispatchVTBL
      :: DISPIDLookupFun
      -> InvokeAction
      -> IO (ComVTable (IDispatch iid) objState)

createStdDispatchVTBL2
      :: [DispMethod objState]
      -> IO (ComVTable (IDispatch iid) objState)
```

createStdDispatchVTBL is the most primitive of these, creating a IDispatch method table for you, leaving you 'just' with the task of specifying the mapping between method names and DISPIDs, and, most importantly, the IO action which implements the functionality of the different methods you support. The supplied invocation action has the following form,

```
type InvokeAction
   = DISPID
  -> MethodKind
  -> [VARIANT]
  -> objState
  -> IO (Maybe VARIANT)
```

the arguments it expects are,

- The `DISPID` identifying what particular method that's being called.
- A *method kind* (badly named, should be `InvokeKind`, or some such), which lets you distinguish (like `IDispatch` does) between 'normal' method invocation and property accessors.
- A list of `VARIANT` arguments. The `createStdDispatchVTBL` implementation does not try to unmarshal any of the arguments for you, leaving that up to you.
- The current state of the object.

The `InvokeAction` possibly returns a `VARIANT` as result - it should do in case it was invoked as a property 'getter' or the method's last argument had `retval` as one of its attributes.

The two actions you supply to `createStdDispatchVTBL` lets you implement the methods of your `IDispatch` component support at a reasonable level of abstraction. You can layer higher-level abstractions on top of it though to suit your programming taste, which is what `createStdDispatchVTBL2` does with its list of `DispMethods`:

```
createStdDispatchVTBL2
      :: [DispMethod objState]
      -> IO (ComVTable (IDispatch iid) objState)

data DispMethod objState
  = DispMethod {
       disp_method_name :: String,     -- method name
       disp_method_id   :: DISPID,     -- its ...
       disp_method_kind :: MethodKind,
       disp_method_act  :: ([VARIANT] -> objState -> IO (Maybe VARIANT))
    }
```

each method is specified by giving its name, `DISPID` and the Haskell IO action which implements it.

To help with the conversion between `VARIANT` and Haskell values, we make good use of the `Variant` type class defined by the `Automation` library, and provide the following helpers:

```
inArg :: ( Variant a )
      => (a -> [VARIANT] -> objState -> IO b)
      -> [VARIANT]
      -> objState
      -> IO b

inoutArg :: ( Variant a )
```

```
              => ( a -> (a -> IO ()) -> [VARIANT] -> objState -> IO b)
              -> [VARIANT]
              -> objState
              -> IO b

outArg :: ( Variant a )
       => ((a -> IO ()) -> [VARIANT] -> objState -> IO b)
       -> [VARIANT]
       -> objState
       -> IO b

retVal :: ( Variant a )
        => ((a -> IO ()) -> [VARIANT] -> objState -> IO (Maybe VARIANT))
        -> [VARIANT]
        -> objState
        -> IO (Maybe VARIANT)

apply_0 :: (objState -> IO ())
        -> [VARIANT]
        -> objState
        -> IO (Maybe VARIANT)

apply_1 :: (Variant a)
        => (objState -> IO a)
        -> (a -> IO ())
        -> [VARIANT]
        -> objState
        -> IO (Maybe VARIANT)

apply_2 :: (Variant a0, Variant a1)
        => (objState -> IO (a0,a1))
        -> (a0 -> IO ())
        -> (a1 -> IO ())
        -> [VARIANT]
        -> objState
        -> IO (Maybe VARIANT)
..etc..
```

To demonstrate their use, suppose we want to supply a Haskell implementation for the following method:

```
[id(2)]HRESULT RegisterTask([in]BSTR s,[out,retval]int* i);
```

The Haskell method will have the type signature

```
registerTask :: String -> SomeObjState -> IO Int
```

to wrap this up in a form that can be included in a `DispMethod`, we define a little wrapper:

```
wrap_registerTask =
  inArg   $ \ s      ->
  retVal  $ \ ret_i  ->
  apply_1 (registerTask s) ret_i
```

The wrapping up of methods inside `DispMethod` values can be simplified further by providing a family of (overloaded) wrappers, grouped by the number of `[in]` and `[out]` arguments their underlying methods expect:

```
dispmethod_0_0 :: String -> DISPID -> (objState -> IO ()) -> DispMethod objState
dispmethod_0_0 name id f = DispMethod name id Method (apply_0 f)

dispmethod_1_0 :: (Variant a)
               => String
               -> DISPID
               -> (a -> objState -> IO ())
               -> DispMethod objState
...
dispmethod_1_2 :: (Variant a0,Variant a1, Variant a2)
               => String
               -> DISPID
               -> (a0 -> objState -> IO (a1,a2))
               -> DispMethod objState
...
```

As a convenience function, `StdDispatch` also exports `createStdDispatch` which calls `createStdDispatchVTBL2` to create the method table and then just creates an interface pointer which supports a singular interface (in addition to `IDispatch` and `IUnknown`, of course):

```
createStdDispatch :: objState
                  -> [DispMethod objState]
                  -> IID iid
                  -> IO (IUnknown iid)
```

Useful when you want to dynamically create and dish out interface pointers, e.g., when creating event sinks.

7.7 The `EnumInterface` library

The `EnumInterface` assists in the creation of COM-style iterators, that is, `IEnum*` interfaces which all support the following set of methods,

```
interface IEnum<It> : IUnknown {
    HRESULT Next
       ( [in] ULONG num
       , [ out
         , size_is(num)
         , length_is(*numFetched)
         ] <It>** outP
       , [out] ULONG *numFetched
       );
    HRESULT Skip  ( [in] ULONG num );
    HRESULT Reset ( void );
    HRESULT Clone ( [out] <It>** ppv );
};
```

They collectively allow you to (repeatedly) iterate over a sequence of elements (of type `It` here).

However, since IDL doesn't allow you to parameterise over the element type you're iterating over, you end up having to define one `IEnum` interface per type.

Not much we can do about that, but at least `EnumInterface` provides you with a generic Haskell implementation of such a family of interfaces:

```
mkEnumInterface :: [a]
                -> Int
                -> (Ptr a -> a -> IO ())
                -> IO (ComVTable iid objState)
```

The `mkEnumInterface` action creates a method table containing the above four `IEnum` methods. It takes the following arguments:

- The list of values to iterate over.
- The size of the marshaled/external representation of an element.
- The by-reference marshaler for the element type.

Here's an example of its use:

```
enumInts :: IO (IEnumInt ())
enumInts = do
  vtbl <- mkEnumInterface [1..] HDirect.sizeofInt (HDirect.writeInt)
  createComInstance "" () [mkIface iidIEnumInt vtbl] iidIEnumInt

type IEnumInt  a = IUnknown (IEnumInt_ a)
dats IEnumInt_ a = IEnumInt__

iidIEnumInt :: IID (IEnumInt ())
iidIEnumInt = mkIID "{...}"
```

7.8 Wrapping up COM components

OK, you've written the IDL, generated the HaskellDirect proxies and implemented the methods of your Haskell COM component - now how do you actually wrap it up as a real, live COM component that's accessable by any old COM client? This section shows you how to do just that, presenting the mechanism by which you declare what components your code implements. The support libraries that take care of the rest are presented in the tailend of this section, but the details of which are probably only of interest to the implementor(s) of this Haskell COM framework.

7.8.1 Declaring your components

By convention, you declare the components that your Haskell code implements by defining `comComponents` in module `Main`:

```
Main.comComponents :: [ComponentInfo]
```

i.e., a COM server can implement one or more components. The `ComponentInfo` is an abstract type containing the various pieces of information that you need to communicate to the underlying support code, so that it can register and create instances of the component for you:

```
data ComponentInfo = .. -- abstract

type RegisterComponent
  =  ComponentInfo
  -> String        -- path to where the component's DLL is installed.
```

```
  -> Bool          -- register or unregister?
  -> IO ()

type ComponentConstructor
  =  String            -- path to where the component's DLL is installed.
  -> IID (IUnknown ()) -- IID of interface to create the component at.
  -> IO (IUnknown ())  -- interface pointer to use (and abuse?)

mkComponentInfo :: CLSID
                -> RegisterComponent
                -> ComponentConstructor
                -> ComponentInfo
```

`ComponentInfo`s are initially built using `mkComponentInfo`, supplying the `CLSID` which identifies the component along with a pair of actions that perform the registration and creation of a component instance.

HaskellDirect will for each component that you declare in IDL (using the `coclass` construct), generate a corresponding `ComponentInfo` for you. See Section ToDo for details of exactly what the generated `ComponentInfo` contains.

You can customise the automatically generated `ComponentInfo` using the following combinators:

```
withProgID         :: ProgID -> ComponentInfo -> ComponentInfo
withVerIndepProgID :: ProgID -> ComponentInfo -> ComponentInfo
withComponentName  :: String -> ComponentInfo -> ComponentInfo

onRegister         :: RegisterComponent -> ComponentInfo -> ComponentInfo
onFinalize         :: IO ()  -> ComponentInfo -> ComponentInfo
```

`withProgID` lets you specify the `ProgID` (i.e., a humanly-readable identifier for your component of the form `<Vendor>.<Component>`), that's sometimes easier to use and refer to (cf. `CreateObject`).

`withComponentName` lets you attach a short descriptive string of what your component is – this string is displayed by some object browsers.

Here's an example use of these combinators:

```
comComponents =
  [ withProgID          "Haskell.PBX.1" $
    withVerIndepProgID  "Haskell.PBX"   $
    withComponentName   "A simple phone directory" $
    PBXProxy.componentInfo --  automatically generated by HDirect.
  ]
```

Notice that the above customisations are entirely optional - your component will be eminently useable if you decide to just use the automatically generated `ComponentInfo`.

The pre-generated `ComponentInfo` includes IO actions which will take care of (un)registering your component in standard ways (i.e., adding itself directly to the Registry.) Should you want to provide custom registration actions via `onRegister`, `ComDll` also exports actions for adding and deleting values from the Registry:

```
regAddEntry :: RegHive
            -> String
            -> Maybe String -- Nothing => new key.
            -> IO ()
regRemoveEntry :: RegHive
```

```
                -> String
                -> String
                -> Bool
                -> IO ()

data RegHive
 = HKEY_CLASSES_ROOT
 | HKEY_CURRENT_USER
 | HKEY_LOCAL_MACHINE
 | HKEY_USERS
 | KHEY_CURRENT_CONFIG
   deriving ( Eq, Ord, Enum )
```

That's it, really - see the example section for instructions of what commands to use to compile up and link the DLL containing the COM server.

7.8.2 In-proc COM server basics

[Starting with this section, we present the gory details of how the COM component wrapper libraries takes a list of `ComponentInfos` and wraps those up as COM in-proc servers. Feel free to ignore (unless you're a sucker for low-level details :-)]

A self-registering, COM in-proc server is delivered in DLL-form, and needs to export the following five entry points:

```
BOOL
STDCALL
DllMain(               // return TRUE is everything is ok.
  HINSTANCE hinstDLL,  // handle to DLL module
  DWORD fdwReason,     // 4 reasons for calling this function,
                       // (attach/detach process/thread)
  void* lpvReserved    // reserved (duh!)
);

HRESULT STDCALL DllCanUnloadNow();
HRESULT STDCALL DllRegisterServer();
HRESULT STDCALL DllUnregisterServer();
HRESULT STDCALL DllGetClassObject(
    REFCLSID rclsid, //CLSID for the class object
    REFIID riid,     //Reference to the identifier of the interface
                     // that communicates with the class object
                     // (normally IID_IClassFactory)
    void** ppv       //Address of output variable that receives the
                     // interface pointer requested in riid
    );
```

where `DllMain` is the main DLL entry point (doh!), which is called upon DLL load (and unload) etc. Registration is done via `DllRegisterServer()` and `DllUnregisterServer()`, whereas component instances are created via `DllGetClassObject()`. More about all this next.

7.8.3 The outermost layer

Almost all of the code required to implement Haskell COM in-proc servers is written in Haskell. The only bit that's not, is a boilerplate C implementation of the signatures given in the previous section:

```
ComDll* comDll = NULL;

extern ComDll* newComDll ( HANDLE hMod );

BOOL
STDCALL
DllMain
   ( HANDLE hModule
   , DWORD reason
   , void* reserved
   )
{
  if (reason == DLL_PROCESS_ATTACH) {
      comDll = newComDll(hModule);
      return TRUE;
  } else {
    if (comDll && reason == DLL_PROCESS_DETACH) {
        (comDll)->dllUnload();
    }
    return TRUE;
  }
}

HRESULT
STDCALL
DllCanUnloadNow (void)
{
  if (comDll) {
     return (comDll)->dllCanUnloadNow();
  } else {
     return S_OK;
  }
}

HRESULT
STDCALL
DllRegisterServer (void)
{
  if (comDll) {
     return (comDll)->dllRegisterServer();
  } else {
    return E_FAIL;
  }
}

HRESULT
STDCALL
DllUnregisterServer (void)
{
  if (comDll) {
    return (comDll)->dllUnregisterServer();
  } else {
    return E_FAIL;
  }
}
```

```
HRESULT
STDCALL
DllGetClassObject
  ( CLSID* rclsid
  , IID*   riid
  , void** ppv
  )
{
  HRESULT hr;
  if (comDll) {
     hr = (comDll)->dllGetClassObject(rclsid, riid, ppv);
     return S_OK;
  } else {
     return E_FAIL;
  }
}
```

As is evident, the wrappers simply delegate to methods in a `ComDll` structure:

```
typedef struct IComDll {
   void    (*dllUnload)();
   HRESULT (*dllCanUnloadNow)();
   HRESULT (*dllRegisterServer)();
   HRESULT (*dllUnregisterServer)();
   HRESULT (*dllGetClassObject)(CLSID* clsid, IID* iid,
                                void** ppv);
} ComDll;

extern ComDll* newComDll(HANDLE hMod);
```

which is created via `newComDll`. Unsurprisingly, `newComDll` is implemented in Haskell, and its implementation is presented in the next section.

7.8.4 Implementing the DLL entry points in Haskell

Implementing the DLL entry points of a COM in-proc server in Haskell means that we need to be able to wrap up Haskell functions behind a C-callable interface in order to fill in a `ComDll` method table.

We show how to do this here using the GHC/Hugs FFI, which is only supported fully under GHC at the moment. Getting this to work under Hugs / HaskellScript isn't beyond the realms of possibility either, but currently un-supported (by HaskellDirect, at least).

The `ComDll` module takes care of implementing the goods, with the `ComDll` method table constructor being:

```
createIComDll :: Addr{-HMODULE-}
              -> [ComponentInfo]
              -> IO (VTable iid_comDllState ComDllState)
```

In addition to the Win32 module `HANDLE`, it is passed the user-supplied list of `ComponentInfo` of the components to wrap up. The connection between `createIComDll` and the user's `comComponents` is made by `newComDll` in a separate module, `ComDllMain`:

```
module ComDllMain where
```

```
import Main   ( comComponents )
import ComDll ( createIComDll )
import Com    ( putMessage )
import Addr

newComDll :: Addr -> IO Addr
newComDll handle = createIComDll handle comComponents

foreign export ccall "newComDll" newComDll :: Addr -> IO Addr
```

The reason for putting `newComDll` in a separate module is that we want to share the code contained in the `ComDll` library between all our COM in-proc servers. So, `ComDllMain` just makes the connection between it and the user's `Main` module, and has to be included when linking the component DLL.

The definition of `newComDll` makes use of a `foreign export` FFI declaration in order to make it callable from C code. When the component DLL is loaded, the entry point `DllMain()` is invoked, which then calls `newComDll`.

Returning to `createIComDll`,

```
createIComDll :: Addr{-HMODULE-}
              -> [ComponentInfo]
              -> IO (VTable iid_comDllState ComDllState)
createIComDll hMod components = do
   state       <- newComDllState hMod components
   meths       <- iComDllEntryPoints state
   createVTable meths

iComDllEntryPoints :: ComDllState -> IO [Addr]
iComDllEntryPoints state = do
  addrOf_DllUnload           <- export_DllUnload   (dllUnload state)
  addrOf_DllCanUnloadNow     <- export_nullaryMeth (dllCanUnloadNow state)
  addrOf_DllRegisterServer   <- export_nullaryMeth (dllRegisterServer state)
  addrOf_DllUnregisterServer <- export_nullaryMeth (dllUnregisterServer state)
  addrOf_DllGetClassObject   <- export_dllGetClassObject (dllGetClassObject state)
  return [ addrOf_DllUnload
         , addrOf_DllCanUnloadNow
         , addrOf_DllRegisterServer
         , addrOf_DllUnregisterServer
         , addrOf_DllGetClassObject
         ]
```

it fills in a `ComDll` method table by exporting five Haskell functions using `foreign export dynamic` FFI declarations,

```
foreign export ccall dynamic
   export_DllUnload :: (IO ()) -> IO Addr
foreign export ccall dynamic
   export_nullaryMeth :: (IO HRESULT) -> IO Addr

foreign export ccall dynamic
   export_dllGetClassObject :: (Addr -> Addr -> Addr -> IO HRESULT) -> IO Addr
```

From the declaration for `export_dllGetClassObject`, the Haskell system will generate an IO action with its stated type. When applied to an IO action, it will return a C function pointer that when it is invoked, will execute the IO action, passing it the arguments that it was passed on the C stack.

In our case, the actions that we export are partially applied to the state that each component wrapper needs to maintain:

```
data ComDllState
  = ComDllState {
      dllPath    :: String,
      components :: IORef [ComponentInfo],
      lockCount  :: IORef Int
     }

newComDllState :: Addr{-HANDLE-}
               -> [ComponentInfo]
               -> IO ComDllState
```

Of the entries in the `ComDll` method table, the `dllGetClassObject` method is the one that's really of interest:[4]

```
dllGetClassObject :: ComDllState
                  -> Ptr CLSID
                  -> Ptr (IID a)
                  -> Ptr (PrimIP a)
                  -> IO HRESULT
dllGetClassObject comDll rclsid riid ppvObject = do
  iid <- unmarshallIID riid
  if ( iidToGUID iid /= iidToGUID iidIClassFactory ) then
     return e_NOINTERFACE
   else do
    clsid <- unmarshallCLSID rclsid
    cs    <- readIORef (components comDll)
    case lookupCLSID clsid cs of
      Nothing -> return cLASS_E_CLASSNOTAVAILABLE
      Just i  -> do
         ip <- createClassFactory (newInstance i (dllPath comDll))
         writeAddr ppvObject (ifaceToAddr ip)
         return s_OK

lookupCLSID :: CLSID -> [ComponentInfo] -> Maybe ComponentInfo
```

The `DllGetClassObject()` method is eventually called by COM on behalf of a client wanting to create an instance of your component.[5] The `riid` argument holds the IID of the *component constructor* (the class *object* or *factory* using COM-speak). In principle, you should be able to use any old interface for this, but at the moment, we only support `IClassFactory`.

Assuming the `rclsid` argument to `dllGetClassObject` identifies a component that's supported, we create an instance of a *class factory* through which a client can create instances of your component. To be able to do just that, the class factory constructor needs to be passed the Haskell action that actually instantiates the component:

[4]Consult the source code for `ComDll` if you want to find out more about how the others are implemented.

[5]Consult other COM documentation (the COM spec, for instance) for details of how COM maps from component identity (i.e., `CLSID`s) to your in-proc DLL.

```
createClassFactory :: (IID (IUnknown ()) -> IO (IUnknown ()))
                   -> IO (IClassFactory ())
```

The next section shows how this class factory is implemented.

7.8.5 A class factory

The `IClassFactory` interface is defined as follows:

```
[...]
interface IClassFactory : IUnknown {
 HRESULT CreateInstance
              ( [in]IUnknown* pUnkOuter
              , [in]REFIID    riid
              , [out]void**   ppv
              );
 HRESULT LockServer
              ( [in]BOOL fLock );
};
```

`LockServer` gives you a way of keeping a class factory in memory, and not have it unload itself – an optimisation that's occasionally worthwhile. `CreateInstance` does the actual creation of a component,

```
createInstance :: This_ClassFactory
              -> Ptr (PrimIP a)
              -> Ptr (IID (IUnknown b))
              -> Ptr (PrimIP b)
              -> IO HRESULT
createInstance this punkOuter riid ppv
 | punkOuter /= nullAddr = return cLASS_E_NOAGGREGATION
 | otherwise             = do
    st   <-  getObjState this
    iid  <-  unmarshallIID riid
    unk  <-  (new_instance st) iid
    writeAddr ppv (ifaceToAddr unk)
    return s_OK
```

As you can see, there's no support for aggregation just yet (but there ought to be). Non-aggregated instances are created by simply delegating to the action that was supplied when creating the class factory. The object state that's accessible from an `IClassFactory` interface pointer has got the following type:

```
data ClassFactory a
 = ClassFactory {
        new_instance :: (IID (IUnknown ()) -> IO (IUnknown ())),
        lockCount    :: (IORef Int)
   }

type IClassFactory a   = IUnknown (ClassFactory a)
type This_ClassFactory = Ptr (IClassFactory ())
```

8 IDL resources

The user's manual makes no attempt at precisely specifying IDL and all its features. Instead, we will in this section give some pointers to further information should you want to understand all the nuts and bolts of IDL:

- The DCE IDL *specification* `<http://www.opengroup.org/pubs/catalog/c706.htm>` is available on-line from the OpenGroup. It is free, but to view it, you'll have to supply your name and e-mail address.
- The OMG CORBA/IIOP 2.2 specification includes a a *chapter* `<http://www.omg.org/corba/cichpter.htm>` on the IDL used to describe CORBA interfaces and objects.
- A recent issue of the Microsoft Systems Journal (Aug 1998), contains an *IDL tutorial* `<http://www.microsoft.com/msj/0898/idl/idl.htm>` by Bill Hludzinski.
- The *Microsoft Developer Network* `<http://www.microsoft.com/msdn/>` site contains a wealth of information on COM and technologies that builds on this component model. Search for "MIDL" to find the Microsoft IDL documentation.
- The definite reference to COM is its *specification* `<http://www.microsoft.com/com/comdocs.asp>`, which explains quite succinctly the features and principles behind COM. Well worth a look.
- The *HaskellScript home page* `<http://www.haskell.org/active/activehaskell.html>` contains links to a number of papers on using COM components from Haskell, as well as links to HaskellScript.
- *COM IDL and Interface Design* by A. Major, Wrox Press, covers IDL in some detail.
- A wealth of other COM literature..

If you have other IDL resources that you have found useful, let us know, and we'll extend the above list.

9 Feedback/reporting bugs

To report your experiences of using HaskellDirect to generate correct (or incorrect, as the case might be :-) Haskell output, send mail to `glasgow-haskell-bugs@haskell.org`, explaining what happened.

Same goes for the contents of this manual - if you found parts wrong/confusing/misleading/incomplete or there were issues that you wished had been covered in more detail, we'd really like to hear about it.