Calling hell from heaven and heaven from hell

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

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

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

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

Abstract

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

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

1 Introduction

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

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

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

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

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

2 Overview

We begin by giving an overview of our architecture (Figure 1). A Haskell program (grey box) that implements a

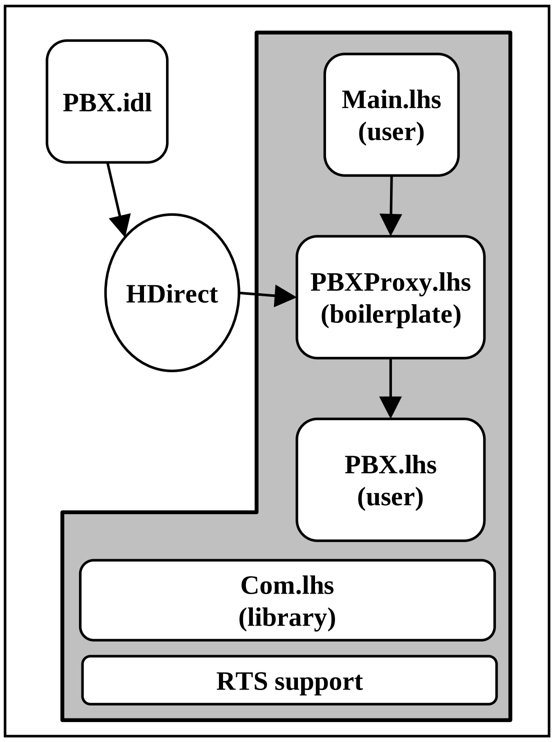

Figure 1: A COM component in Haskell

COM component consists of three parts:

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

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

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

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

3 The foreign function interface

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

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

3.1 Foreign static import and export

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

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

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

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

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

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

3.2 Variations on the theme

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Stable pointers and foreign objects

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

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

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

3.4 Dynamic import

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

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

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

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

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

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

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

3.5 Dynamic export

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

[2] Dynamically Linked Library

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

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

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

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

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

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

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

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

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

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

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

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

[4] The Appendix introduces `IORefs`.

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

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

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

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

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

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

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

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

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

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

3.6 Implementing dynamic export

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

Our implementation for the Glasgow Haskell Compiler works by taking advantage of the *static* version of `foreign export`. Here, for example, is how we implement `mkWndEnumProc`. We repeat its declaration here:

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

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

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

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

- register the Haskell function as a stable pointer;
- dynamically generate a code fragment;
- return the address of this dynamically generated code.

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

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

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

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

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

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

3.7 Related work

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

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

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

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

4 How COM works

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

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

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

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

The procedure `CoCreateInstance` is best thought of as an operating system procedure. (In real life, it takes more parameters than those given above, but they are unimportant

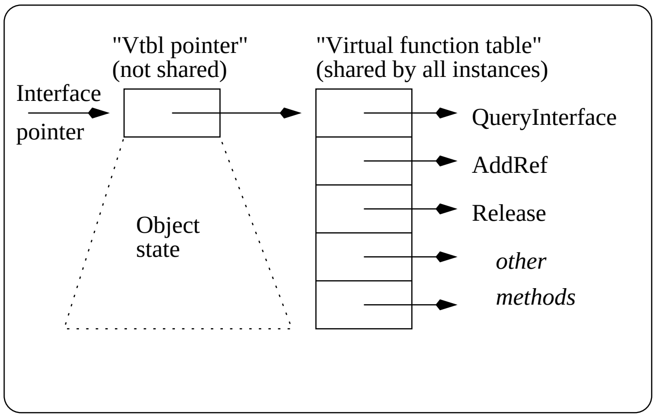

Figure 2: Interface pointers

here.) Calling `CoCreateInstance` creates an instance of an object whose *class identifier*, or CLSID, is held in `cls_id`. The class identifier is a 128-bit *globally unique identifier*, or GUID. Here "globally unique" means that the GUID is a name for the class that will not (ever) be re-used for any other purpose anywhere on the planet. A standard utility allows an unlimited supply of fresh GUIDs to be generated locally, based on the machine's IP address and the date and time.

The code for the class is found indirectly via the *system registry*, which is held in a fixed place in the file system. This double indirection of CLSIDs and registry makes the client code independent of the specific location of the code for the class. Next, `CoCreateInstance` loads the class code into the current process (unless it has already been loaded). Alternatively, one can ask COM to create a new process (either local or remote) to run the instance.

4.1 Interfaces and method invocation

A COM object supports one or more *interfaces*, each of which has its own globally-unique *interface identifier* or IID. That is why `CoCreateInstance` takes a second parameter, `iface_id`, the IID of the desired interface; `CoCreateInstance` returns the *interface pointer* of this interface in `iface_ptr`. There is no such thing as an "object pointer", or "object identifier"; there are only interface pointers.

The IID of an interface uniquely identifies the complete *signature* of that interface; that is, what methods the interface has (including what order they appear in), their calling convention, what arguments they take, and what results they return. If we want to change the signature of an interface, we must give the new interface a different IID from the old one. That way, when a client asks for an interface with a particular IID, it knows exactly what that interface provides.

A COM interface pointer is (deep breath) a pointer to a pointer to a table of method addresses (Figure 2). Notice the double indirection, which allows the table of method addresses to be shared among all instances of the class. Data specific to a particular instance of the class, notably the object's state, can be stored at some fixed offset from the "vtbl pointer" (Figure 2). *The format of this information is entirely up to the object's implementation; the client knows nothing about it.* Lastly, when a method is invoked, the interface pointer must be passed as the first argument, so that the method code can access the instance-specific state. Taking all these points together, we can now see why a method invocation looks like this:

```
(*iface_ptr)[3]( iface_ptr, x, y, z );
```

None of this is language specific. That is, COM is a binary interface standard. Provided the code that creates an object instance returns an interface pointer that points to the structures just described, the client will be happy. In theory, the parameter passing conventions for each method can be different (but fixed in advance). In practice, they match the `__stdcall` convention used by C and C++.

Interface pointers provide the sole way in which one can interact with a COM object. This restriction makes it possible to implement *location transparency* (a major COM war-cry), whereby an object's client interacts with the object in the same way regardless of whether or not the object is in the same address space, or even in the same machine, as the client. All that is necessary is to build a *proxy* interface pointer, that *does* point into the client's address space, but whose methods are stub procedures that marshal the data to and from across the border to the remote object.

4.2 Getting other interfaces

A single COM object can support more than one interface. But as we have seen before `CoCreateInstance` returns only one interface pointer. So how do we get the others? Answer: every interface supports the `QueryInterface` method, which maps an IID to an interface pointer for the requested IID or fails if the object does not support the requested interface. So, from any interface pointer (`iface_ptr`) on an object we can get to any other interface pointer (`iface_ptr2`) which that object implements, for example:

```
err_code = (*iface_ptr)[0]( iid2, &iface_ptr2 );
```

Why "`[0]`"? Because `QueryInterface` is at offset `0` in every interface.

The COM specification requires that `QueryInterface` behaves consistently. The `IUnknown` interface on an object is the identity of that object; queries for `IUnknown` from any interface on an object should all return exactly the same interface pointer. Queries for interfaces on the same object should always fail or always succeed. Thus, the call `(*iface_ptr)[0](iid2,&iface_ptr2);` should not succeed at one point, but fail at another. Finally, the set of interfaces on an object should form an equivalence relation.

4.3 Reference counting

Each object keeps a *reference count* of all the interface pointers it has handed out. When a client discards an interface pointer it should call the `Release` method *via* that interface pointer; every interface supports the `Release` method. Similarly, when it duplicates an interface pointer it holds, the client should call the `AddRef` method *via* the interface

```
[object,
 uuid(00000000-0000-0000-C000-000000000046),
 pointer_default(unique)
]
interface IUnknown {
 HRESULT QueryInterface( [in] REFID iid,
                         [out] void **ppv );
 ULONG   AddrRef( void );
 ULONG   Release( void );
}

[ object, uuid(...) ]
interface ILookup : IUnknown {
  HRESULT LookupByName   ( [in,string]char* name,
                           [out,string]char** number );
  HRESULT LookupByNumber( [in,string]char* number,
                          [out,string]char** name );
}

[ object, uuid(...) ]
interface IInsert : IUnknown {
  HRESULT Insert( [in,string]char* name,
                  [in,string]char* number );
}

[ uuid(...) ]
coclass PBX {
  [default] interface ILookup;
  interface interface IInsert;
}
```

Figure 3: The IDL for `IUnknown` and `PBX`

pointer; every interface also supports the `AddRef` method. When an object's reference count drops to zero it can commit suicide — but it is up to the object, not the client, to cause this to happen. All the client does is make correct calls to `AddRef` and `Release`.

Every interface supports the three methods `QueryInterface`, `AddRef`, and `Release`. The three together constitute the `IUnknown` interface, which every other interface extends.

4.4 Describing interfaces

Since every IID uniquely identifies the signature of the interface, it is useful to have a common language in which to describe that signature. COM has such a language, called IDL (Interface Definition Language), but IDL is not part of the core COM standard. You do not have to describe an interface using IDL, you can describe it in classical Greek prose if you like. All COM says is that one IID must identify one signature.

Describing an interface in IDL is useful, though, because it is a language that all COM programmers understand. Furthermore, there are tools that read IDL descriptions and produce language-specific declarations and glue code. For example, the Microsoft MIDL compiler can read IDL and produce C++ class declarations that make COM objects look exactly like C++ objects (or Java, or Visual Basic).

As a short example, Figure 3 gives the IDL description of the `IUnknown` interface, the interface that every other extends. The 128 bit long constant is the GUID for the `IUnknown` interface. Also presented are the class and interface declarations for a simple telephone directory component, `PBX`. The `PBX` class supports two interfaces, `ILookup` and `IInsert`. The former has two methods, in addition to the standard `IUnknown` methods, while the latter has one. (The class and interface GUIDs are elided to "`...`" for brevity.)

5 Polymorphism expresses single inheritance

Our earlier paper showed how to create and invoke COM components from Haskell. We found that we were able to make compelling use of polymorphism to offer type security right at the heart of our implementation. Here are the types of the Haskell equivalents of `CoCreateInstance` and `QueryInterface`:

```
coCreateInstance :: CLSID   -> IID iid -> IO (IUnk iid)
queryInterface   :: IID iid -> IUnk a  -> IO (IUnk iid)
```

- `CLSID` is the type of class GUIDs.
- `IID iid` is the type of interface GUIDs, *but parameterised by,* `iid`*, the "interface type".*
- `IUnk iid`[5] is the type of interface pointers, *again parameterised by its interface type.*

The polymorphism in `coCreateInstance` and `queryInterface` elegantly ensures that the interface pointer returned is statically checked to support the same methods as the IID that was passed.

Whenever `coCreateInstance` or `queryInterface` obtains a new interface pointer of type `IUnk iid` from COM, it attaches a finaliser to it (Section 3.3), so that when the Haskell program lets go of the interface pointer, the finaliser will automatically call `Release`. In this way, managing COM object reference counts is invisible to the programmer.

5.1 Interface types

What are these "interface types"? For every interface (GUID) defined in the IDL for a component, H/Direct simply define a fresh Haskell type — the interface type. There is a one-to-one correspondence between interface IDs and interface types, which is why we use "`iid`" for a type variable that ranges over interface types.

Strangely, such an interface type is an abstract data type with no operations, nor do we ever create a value of the type. For example, consider the `ILookup` interface in Figure 3. When fed this IDL, H/Direct will produce a Haskell module containing the following declarations (among others):

```
data ILookupT a = ILookupT
      -- The interface type

type ILookup a = IUnknown (ILookupT a)

iidILookup :: IID (ILookupT ())
iidILookup = newIId "...GUID for ILookup..."
```

[5]In the real implementation it is called "`IUnknown`", but "`IUnk`" made our typesetting easier!

The interface type for `ILookup` is called `ILookupT`. It is declared as an algebraic `data` type with a single constructor[6] We will return shortly to the type parameter for `ILookupT`; just ignore it for now. Next, there is a type synonym, that defines `ILookup a` to be the type of interface pointers for interfaces of type `ILookup`. Finally, a suitably-typed interface ID for `ILookup` is defined.

H/Direct also generates client stub definitions for the methods of the interface:

```
lookupByName   :: String -> ILookup a -> IO String
lookupByNumber :: String -> ILookup a -> IO String
```

Notice that these each take a typed interface pointer as their argument. It is impossible for the Haskell application to accidentally call `lookupByName` passing it an interface pointer to an `IInsert` interface, say. And the *only* way the application can construct an interface pointer of type `ILookup a` is by calling `coCreateInstance` or `queryInterface`!

5.2 Inheritance

Why is `ILookupT` parameterised? Because it is possible to define another interface that extends `ILookup`. The IDL might look like this:

```
interface ISearch : ILookup { ... }
```

Given this, H/Direct will generate the following:

```
data ISearchT a = ISearchT

type ISearch a = ILookup (ISearchT a)

iidISearch :: IID (ISearchT ())
iidISearch = ...
```

Now, the beautiful thing is this: if I have an interface pointer of type `ISearch t`, then I can use `lookupByName` on it. Why? Because

$$\texttt{ISearch t} = \texttt{ILookup (ISearchT t)}$$

(just by expanding the type synonym for `ISearch`). That is, *every interface pointer for* `ISearch` *is automatically an interface pointer for* `ILookup`*, and indeed also an interface pointer for* `IUnknown`.

Now we can understand the type of `iidISearch` as well:

```
iidISearch :: IID (ISearchT ())
```

`iidISearch` is the interface ID for `ISearch` *exactly*, expressed by instantiating the type parameter to `()`.

In short, we have been able to use simple polymorphic instantiation to model (single, interface) inheritance. In retrospect the idea is quite obvious, and doubtless has been invented many times before, but we have been unable to find a published account.

[6] It would be better to declare it as a type with no constructors, since we never use the constructor, but Haskell does not allow that.

6 Encapsulating Haskell as a COM component

Next, we turn to our third main theme, the task of implementing a specified COM component in Haskell. The starting point is an IDL specification for the interface(s) the component must offer; as a running example we use the telephone directory given in Figure 3. This is closely based on the example used in [4] to introduce Component Pascal's support for interacting with COM.

We tackle the encapsulation in three clearly-separated "layers" (Figure 1):

- Code written by the application programmer writes (Section 6.1). There are two things to do here: provide an implementation of the component, and register it with COM so that other components can invoke it.
- Code generated by H/Direct from the `PBX` IDL (Section 6.2). This boilerplate code deals with marshalling arguments between Haskell and the client; it also deals with creating the component's vector tables and interface pointers in exactly the form expected by COM clients.
- Fixed code that lives in the `Com` library (Section 7).

6.1 The programmer's eye view

What does the Haskell programmer have to do to implement `PBX` in Haskell? First he feeds the IDL to H/Direct, which generates a Haskell module `PBXProxy.hs` (Figure 1). This module imports a Haskell module `PBX.lhs`, which provides the programmer's implementation of the `PBX` functionality. H/Direct optionally outputs a skeleton for this module, but the programmer must complete it by providing:

- A type declaration for the state of the `PBX` object. This type is given the same name as the class. For example:

  ```
  type Name   = String
  type Number = String
  type PBX    = IORef [(Name,Number)]
  ```

  Here the state `PBX` is held in a mutable `IORef` cell.
- A initialiser, `initPBX`, for the `PBX` state. For example:

  ```
  initPBX :: IO PBX
  initPBX = newIORef []
  ```
- An implementation for each method. The Haskell type of each method is derived from the corresponding IDL type; this type translation is given in detail in [2]. For example:

  ```
  lookupByName :: String -> PBX -> IO String
  lookupByName want pbx
     = do { pairs <- readIORef pbx
          ; case [num | (nm,num) <- pairs,
                        nm == want]
            of
             (n:_) -> return n
             []    -> coError E_Fail
          }
  ```

The last parameter of each method is the state of the object. The function `coError` raises an exception in the `IO` monad, passing the `E_Fail` return code, which is marshalled into COM's `E_FAIL` return code by H/Direct.

Finally, the programmer must make the new component known to COM by supplying a main module, `Main.lhs`, as follows:

```
module Main where
  import Com( coRegister )
  import PBXProxy( pbx )

  main :: IO ()
  main = coRegister [pbx]
```

When the Haskell program is run, the call to `coRegister` registers the component(s) defined in its argument list. This step registers Haskell functions through which a client can create instances of the component(s). If a single Haskell program implements more than one COM component, `main` would import several `XProxy` modules, and would have several items in the list passed to `coRegister`.

And that is all the programmer has to do! Next, we look behind the scenes, and study the `PBXProxy` and `Com` modules.

6.2 Creating a component instance

H/Direct generates the module `PBXProxy` from the IDL for `PBX`, which exports the single value `pbx`. This value `pbx` encapsulates the complete implementation of the component[7]

```
pbx :: CoComponent

data CoComponent
  = CoComponent
    { componentCLSID  :: CLSID
    , componentProgID :: String
    , newInstance     :: IID iid -> IO (IUnk iid)
    }
```

A `CoComponent` is a triple of a class ID, a string through which the class ID can be looked up in the registry — clients do not always know the class ID — and a way to create a new instance of the component. The first two fields are easy to generate from the IDL. We will focus on the last, `newInstance`.

To create an instance of a COM component we need to construct an interface pointer that looks precisely as depicted in Figure 2. We represent an interface pointer is a pointer to a `malloc`'d pair of (a) a method vector table pointer and (b) a (stable pointer to) the object's state[8]. All the interfaces for a particular object share a common state. So there are two things we must be able to do:

1. *Create a method vector table.* In the compiled implementation we could do this statically, but that is not possible in the interpreter, so we provide a function that dynamically builds a vector table.

   ```
   type CoVTable iid st = ...

   newCoVTable :: [Addr] -> CoVTable iid st
   ```

   Function `newCoVTable` uses `malloc` to allocate a fixed, never-freed, vector table, returning its address. The `CoVTable` type is parameterised by the interface type (`iid`) of the interface it implements, and object state (`st`) understood by the methods.

   The method addresses passed to `newCoVTable` point to procedures that can be called directly by other COM objects. They can be generated using `foreign export`. `newCoVTable` prefixes this list with three further addresses, for the `IUnknown` methods `QueryInterface`, `AddRef`, and `Release`. (A variant is provided for those who want to write their own implementations of these methods – see Section 7.2.)

2. *Create an instance of the object.* A COM object may support several interfaces, so we must pass a list of (IID,VTable) pairs, each of type `IfaceSpec`:

   ```
   data IfaceSpec st
     = forall iid. IfaceSpec (IID iid)
                             (CoVTable iid st)

   newCoInstance :: st -> [IfaceSpec st]
                 -> IID iid -> IO (IUnk iid)
   ```

   `newCoInstance` takes an initial state, a list of interfaces (each specified as a (IID,VTable) pair), and an IID, and returns a suitable interface pointer to the object.

   The `data` type declaration for `IfaceSpec` uses an experimental extension of Haskell that provides existential data types. The data type has one constructor, `IfaceSpec`, with type:

   ```
   IfaceSpec :: IID iid -> CoVTable iid st
             -> IfaceSpec st
   ```

   The `IID` and `CoVTable` must have compatible `iid` types, but that type does not show up in the type of the constructed value. Hence, a list of `IfaceSpec`s may differ in their `iid`s, but will all have the same `st`. This extension, first suggested by Laufer [7], is implemented by several Haskell compilers, include GHC, `hbc`, and Hugs.

We are finally ready to give the code for the `PBXProxy` module. Remember that its sole export is the component `pbx`.

```
module PBXProxy( pbx ) where

import PBX( PBX, initPBX, lookupByName,
            lookupByNumber, insert )
import Com( CoComponent(..),
            IfaceSpec(..),
            newCoInstance, newCoVTable,
            CoIPRep, getCoState )

pbx :: CoComponent
pbx = CoComponent {
       componentCLSID = "...",
```

[7] The actual data type contains a few extra fields — for example a string giving a short description of the component.

[8] In principle, we could instead create a fresh method table for each instance of the object; the methods could then have the object state as a free variable, just `getTitle` did in Section 3.5. But that would mean much method-table duplication, so instead we follow COM's hint, and use a fixed method table, shared among all instances.

```
        componentName  = "PBX",
        newInstance    = newPBX
      }

newPBX :: IID iid -> IO (IUnk iid)
newPBX = do { init <- initPBX
            ; newCoInstance init pbxSpecs }

pbxISpecs :: [IfaceSpec PBX]
pbxISpecs = [IfaceSpec iidIInsert vtInsert,
             IfaceSpec iidILookup vtLookup]

vtInsert :: VTable IInsertT PBX
vtInsert = newCoVTable [wrapInsert]

vtLookup :: VTable ILookupT PBX
vtLookup = newCoVTable [wrapLookupByName,
                        wrapLookupByNumber]

-- Definitions of IInsertT, iidIInsert etc as before

foreign export "LookupByName" wrapLookupByName
  :: CoIPRep PBX -> Addr -> Addr -> IO ()

wrapLookupByName ip p_name p_number
  = do { st     <- getCoState ip ;
         name   <- unmarshallString p_name ;
         number <- insert name st ;
         writeString p_number number }

-- Similar wrappers for LookupByName, LookupByNumber
```

All of this code is generated from the PBX IDL by H/Direct, which is a good thing, because it is quite tiresome to write.

The definitions of `pbx`, `newPBX` and `pbxISpecs` are straightforward. The vector tables, `vtInsert` and `vtLookup`, are allocated on demand, by `newCoVTable`. The addresses in the vector table are obtained using `foreign export`. The function thus exported is a wrapper function (`wrapInsert` is an example) that takes the raw "self" interface pointer as an argument. The purpose of this interface pointer is to get the object state, so we give it the type `CoIPRep PBX`, and provide the operation:

```
getCoState :: CoIPRep st -> IO st
```

which extracts the state component from an interface pointer. Now we can pass that state on to the user-written method `insert`, imported from module `PBX`.

It may seem strange that in Section 5 we gave interface pointers a type (`IUnk`) parameterised by an *interface type*, while here we parameterise a different type (`CoIPRep`) by the object *state*. How peculiar! However, even though both are *represented* by a single address, they play quite different roles. A value of type `IUnk iid` is a *client-side* interface pointer for an object held elsewhere; its state is invisible, and when it is finalised (Section 3.3) we must call its `Release` method. In contrast, a value of type `CoIPRep st` is a server-side interface pointer; its state is visible (because the 'this' pointer is passed to the method implementation), and when there are no further references we need only call `free` to return the store to `malloc`.

When H/Direct is generating code for Hugs, it can only use `foreign export dynamic`, so the code for `vtInsert` and `vtLookup` is a little more indirect, but still straightforward.

7 The `Com` library

The bottom layer of the encapsulation is the fixed, generic Haskell library `Com.lhs` to support COM objects. We do not have space to present detailed code; instead we summarise what the implementation (completely written in Haskell) does.

7.1 Activation

When COM tries to create an instance of a given class ID, it looks in the registry to find what DLL to activate. Assuming that the DLL has not already been loaded, COM will load it and invoke its initialisation procedure. If the DLL holds a Haskell program, this initialisation procedure runs the Haskell programs function `main`. As indicated in Section 6.1, `main` in turn calls `coRegister`, passing it a list of all the components that the program implements:

```
coRegister :: [CoComponent] -> IO ()
```

`coRegister` can now create a "class factory" by applying the `newClassFactory` function to the list of components:

```
newClassFactory :: [CoComponent] -> CLSID
                -> IID iid -> IO (IUnk iid)
```

All `newClassFactory` has to do is to search the list of components for one with a matching CLSID, and use the corresponding `newInstance` function. A simplified implementation of `coRegister` is:

```
coRegister compts
  = do { cf <- exportCF (newClassFactory compts)
       ; exportClassFactory cf
       }

foreign export dynamic
  exportCF :: (CLSID -> IID iid -> IO (IUnk iid))
           -> IO Addr
```

We use `foreign export dyanamic` to capture the class factory function, and return it to the Haskell initialisation procedure by calling `exportClassFactory` (which does something like set a global variable).

Actually, that is not quite all that `coRegister` has to do. It must also provide functions to deal with the standard COM registration bureaucracy; the details are not important here. The important bit is that `coRegister` exports a class factory through which the outside world can now (finally!) create instances of our Haskell COM component.

7.2 The `IUnknown` interface

In Section 6.2 we said that `newCoVTable` and `newCoInstance` worked together to provide implementation of the `IUnknown` methods, `QueryInterface`, `AddRef`, and `Release`. In this section we outline how this is done.

The basic idea is simple enough. Recall that we represent an interface pointer by a `malloc`'d pair of a pointer to the method vector table, and (a stable pointer to) the Haskell state for the object. For COM objects that use the `Com` library support, the object state is a (Haskell) pair of two values: the "user" state (PBX in the above example), and the "system" state. The system state in turn is a pair of (a) a reference count for the whole object, and (b) a mapping from an IID to an interface pointer:

```
type CoState st = (IORef Int,
                   FiniteMap (IID ()) (CoIPRep st))

type Iface st    = (IID (), CoIPRep st)
```

With this object state in mind, we can provide standard `AddRef` and `Release` methods. They simply adjust the reference count held in the `CoState`. When the reference count drops to zero, `Release` simply frees the stable pointer that keeps the object's state alive. That, in turn, may cause a number of finalizers to get, see Section 3.3.

The `QueryInterface` method uses the IID-to-interface-pointer mapping to do its work. The typing of the mapping looks strange, for two reasons. First, the Haskell type system cannot express the idea of a mapping in which the argument *value* determines the result *type*. One needs dependent types for that. Second, the result of `QueryInterface` is in any case returned immediately to the external client, so little is gained by a sophisticated typing.

With this in mind, `newCoVTable` uses the even-more-primitive `newVTable` to do its work:

```
newVTable :: [Addr] -> VTable iid st
      -- Simply calls malloc

type CoVTable iid st = VTable iid (CoState st, st)
newtype VTable iid st = VTable Addr
```

The function `newCoVTable` prepends the standard implementations for `QueryInterface`, `AddRef`, and `Release`, before calling `newVTable`.

Finally `newCoInstance` allocates an interface pointer for each interface[9] builds the finite mapping from IIDs to interface pointers, and then constructs the object state. Oh! We need the object state before we can construct the interface pointer! So the whole thing has to be wrapped in a `fixIO` knot-tying combinator.

8 Related work

Haskell is not the only advanced programming language to provide a mapping to COM. The Harlequin Dylan system [3] provides a well-engineered COM component framework for Dylan, letting the programmer both create and use COM components. Equipped with such powers, Harlequin Dylan also provides a framework for writing ActiveX controls, something we have yet to tackle. Component Pascal [4] and Microsoft's implementation of Java [9] are two other examples of garbage collected languages which have been integrated with COM.

Component integration is becoming more widespread for functional languages too. Mercury has a CORBA interface [6]. A few days before we submitted this paper, Leroy released a version of Caml that supports COM components, using an architecture similar to that described in this paper.

[9] Incidentally, it is easy to do this lazily, getting the behaviour of *tear-off* interfaces for free [1].

Figure 4: Visual Basic PBX client

We understand that work is in progress on a COM interface for Standard ML of New Jersey.

9 Conclusions

The main burden of this paper has been to give the details of an encapsulation of a Haskell as a COM component.

It is worth stressing that Joe Programmer need to know little of this. All s/he need do is feed the IDL for the component to H/Direct and write the application code. All the details of component construction, reference counting, interface querying, and simple finalisation (such as calling `Release` on interface pointers held by the object), are handled automatically.

Behind the scenes, though, there are many details to attend to, and we have not even discussed them all. (For example, we omitted many details about object registration and finalisation.) Still, we hope to have conveyed something of the flavour.

We have made good use of Haskell's type system to make application code completely type secure, and H/Direct-generated code largely so. We found some interesting uses of polymorphism in so doing. However, we were not able to make all the `Com` library support code type-secure.

All that we describe is implemented in H/Direct and GHC (though some of the function names may differ). We look forward to exploring the possibilities of writing COM components in Haskell. As a tiny example of this, Figure 4 shows a Visual Basic application using the PBX server given in this paper. Real World - here we come!

References

[1] D. Box. *Essential COM*. Addison Wesley, 1998. ISBN: 0-201-63446-5.

[2] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: a binary foreign language interface for Haskell. In *Proc ACM Sigplan International Conference on Functional Programming (ICFP'98), Balitmore*, pages 153–162. ACM, 1998.

[3] D. Gray, J. Hotchkiss, S. LaForge, A. Shalit, and T. Weinberg. COM simplified: Modern Languages and Microsoft's Component Object Model. *Communications of the ACM*, May 1998.

[4] D. Gruntz and B. Heeb. Direct-To-COM Compiler Provides Garbage Collection for COM Objects. In *Proceedings of the 2nd Component User's Conference (CUC)*, Munich, Germany, July 1997. Available on-line from `http://www.oberon.ch/resources/com/dtc_cuc_paper/index.html`.

[5] L. Huelsbergen. A Portable C Interface for Standard ML of New Jersey. AT&T Bell Laboratories, January 1996.

[6] D. Jeffery, T. Dowd, and Z. Somogyi. MCORBA: a CORBA binding for Mercury. In Gupta, editor, *Practical Applications of Declarative Languages*, pages 211–227. Springer Verlag LNCS 1551, 1999.

[7] K Läufer and M Odersky. An extension of ML with first-class abstract types. In *Workshop on ML and its Applications*. 1992.

[8] X. Leroy. *Interfacing C with Objective Caml*. INRIA, Rocquencourt, France. `http://caml.inria.fr/ocaml/htmlman/`.

[9] Microsoft Corporation. Visual J++. `http://www.microsoft.com/java/`.

[10] S. Peyton Jones, S. Marlow, and C. Elliott. Stretching the storage manager: weak pointers and stable names in Haskell. Technical report, Microsoft Research (submitted to ICFP'99), 1999.

[11] S. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM components in Haskell. In *Proc Fifth International Conference on Software Reuse, Victoria*. IEEE, 1998.

[12] S. Peyton Jones and P. Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston*, pages 71–84. ACM, 1993.

[13] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[14] J. R. Rose and H. Muller. Integrating the Scheme and C Languages. In *Proc ACM 1992 Conference on Lisp and Functional Programming*, pages 247–259, 1992.

[15] M. Serrano. Bigloo User's Manual, 1999. `http://kaolin.unice.fr/~serrano/bigloo/bigloo.html`.

[16] C. Szyperski. *Component Software*. Addison Wesley, 1998.

[17] M. Wallace. *Calling Haskell from C using GreenCard*. `http://www.cs.york.ac.uk/fp/nhc/CcallingHaskell.html`.

Appendix: Input/output in Haskell

In Haskell, a function that has type `Int -> Int`, say, is a function from integers to integers, no more and no less. In particular it cannot perform any input/output. Any function that can perform I/O has a result type of the form `IO` τ. This so-called *monadic I/O* has become the standard way to do input/output in purely functional languages [12]. An I/O performing function can be used in a `do` expression, which serves to sequence such computations. For example:

```
main :: IO ()
main = do { l <- getLine ;
            putStr (reverse l)
       }
```

This program uses two standard functions:

```
getLine :: IO String
putStr  :: String -> IO ()
```

When `main` is performed, the `do` arranges first to perform `getLine`, binding the result, of type `String` to `l`. The it performs `putStr (reverse l)`, which displays the reverse of `l`.

Many of the programs in this paper use mutable cells, similar to ML's `ref` type.

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

A value of type `IORef t` is a reference to a mutable cell holding a value of type `t`. The primitives to allocate, read, and write the cell are all in the `IO` monad. Here is a short example, using Haskell's `do` notation:

```
swap :: IORef a -> IORef a -> IO ()
-- Swap the contents of the two cells
swap aref bref = do { a <- readIORef aref
                    ; b <- readIORef bref
                    ; writeIORef aref b
                    ; writeIORef bref a
                    }
```

A primitive IO action is also provided for tying knots,

```
fixIO :: (a -> IO a) -> IO a
```

which is the fixpoint combinator at the level of IO actions.