Scripting COM components in Haskell

Simon Peyton Jones (simonpj@dcs.gla.ac.uk)
University of Glasgow and Oregon Graduate Institute

Erik Meijer (erik@cs.ruu.nl)
University of Utrecht and Oregon Graduate Institute

Daan Leijen (leijen@wins.uva.nl)
University of Amsterdam and Oregon Graduate Institute

December 15, 1997

Abstract

Designers of advanced languages, such as ML, Prolog, or Haskell, face an uphill struggle to persuade potential users of the merits of their approach. In fact, it has hitherto been impossible to find other than niche applications because (foreign language interfaces notwithstanding) it has been too difficult to integrate software components written in new languages with large bodies of existing code.

Microsoft's Component Object Model (COM) offers this community a new opportunity. Because the interface between objects is by design language independent and arms-length, it is possible either to write glue programs that integrate existing COM objects, or to write software components whose services can be used by clients written in more conventional languages.

We describe our experience of exploiting this opportunity in the purely-functional language Haskell. We describe a design for integrating COM components into Haskell programs, and we demonstrate why someone might want to script their COM components in this way.

This paper has been submitted to Software Reuse 1998.

keywords

Haskell, COM, CORBA, software components, lazy evaluation, functional programming, strong typing, polymorphism, scripting, interoperability, equational reasoning.

1 Introduction

Programming-language researchers have a serious marketing problem. Apart from a relative handful of enthusiasts, our languages are not widely used, because no potential customer is prepared to revolutionize the way they build their systems — and rightly so. Despite some work on foreign-language interfaces, it has been hard to provide an evolutionary path that would enable a potential customer to experiment with a new language at a low level of commitment.

Microsoft's Component Object Model (COM) is a widely-deployed, binary standard for software components [12]. Because its language independence, COM presents two new opportunities for programming-language researchers. COM makes it easier to use a new language either (a) to glue together, or script, a collection of existing COM components to make a larger application or component, or (b) to implement a new COM component that a client can use without knowledge of its implementation language.

We have begun to exploit the first of these opportunities in the context of the purely functional programming language Haskell [4]. In this paper we describe an interface between Haskell and COM that makes it easy to script COM components from a Haskell program. We make two main contributions:

- A graceful and strongly typed accommodation of COM within the host language is important. We present a design for how COM could appear to the Haskell programmer.
- If the exercise is to be more than just "Gosh, we can script COM in Haskell as well as in Visual Basic" then it is important to demonstrate some added value from using a higher-order, typed language. We offer such a demonstration, in the form of a case study.

2 The opportunity

Until recently it has been much easier for a client program to use software components (libraries, classes, abstract data types) written in the same language:

1. The specification of the interface between the component and its clients is usually given in a language-specific way; for example, as C++ class descriptions.

2. The calling convention between client and component is often language-specific, or perhaps even unspecified (because both client and component are assumed to be compiled with the same compiler)

3. Programmers can assume a rather intimate coupling between the address spaces of client and component; for example, the client might pass a pointer into the middle of an array, to be side-effected by the component.

COM encapsulates a software component in a way that contrasts with each of these three aspects:

- The interface between client and component is specified in IDL (COM's Interface Definition Language). For each particular language, tools are provided to convert IDL into the corresponding specification in that language (section 3.4).
- COM specifies the client/component interface at a binary level, independently of any particular language or compiler (section 3.1).
- Parameters are expected to be marshalled from the client's address space to the component's address space, and vice versa. Sometimes the two share an address space, in which case marshalling need do no copying, but all COM calls provide enough information to do such marshalling.
- Interfacing between two languages often carries performance overheads, because of differing data representation and memory-allocation policies. When the alternative is a native-language interface between client and component, these extra overheads can seem rather unattractive.

  However, anyone using COM has already bitten the bullet: they have declared themselves willing to accept a hit in programming convenience, and perhaps a hit in performance (for marshalling), in exchange for the advantages that COM brings.

These are not COM's only advantages. For example, one of the primary motivations for using COM concerns version control and upgrade paths for software components, which we have not mentioned at all so far. However, these additional properties are well described elsewhere, [11, 12, 1, 2, 3] and do not concern us further in this paper, except in so far as they serve as motivators for people to write and use COM components.

Also, COM is not alone in having these properties. Numerous research projects had similar goals, in particular CORBA [13]. In fact, almost everything in the rest of this paper would apply to CORBA as well as COM, because CORBA is largely compatible with COM. We stick to COM for the sake of being concrete (it has a well-defined, mature and stable specification) and because of its widespread use. With more than 200 million systems worldwide using it, COM offers designers of advanced languages the best opportunities for reusing software components.

3 How COM works

Although there are many very fat books about COM (e.g. [12]), the core technology is quite simple, a notable achievement. This section briefly introduces the key ideas. We concentrate exclusively on *how* COM works, rather on *why* it works that deal; the COM literature deals with the latter topic in detail.

Here is, in C, how a client 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 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.

3.1 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 "ob-

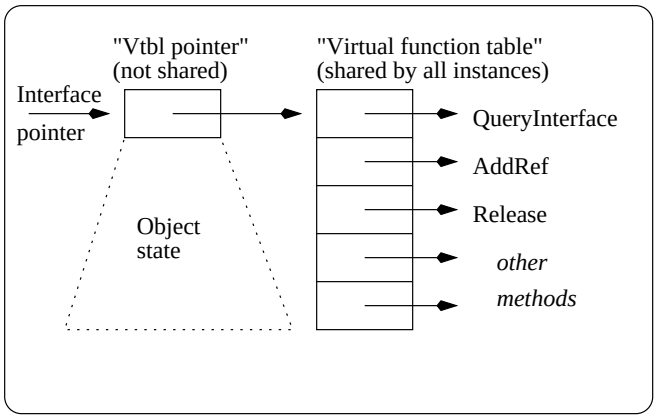

Figure 1: Interface pointers

ject 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 1). 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 1). *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.

3.2 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 `IUnkown` 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.

3.3 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 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`.

3.4 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) [6], 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

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

Figure 2: The `IUnknown` interface in IDL

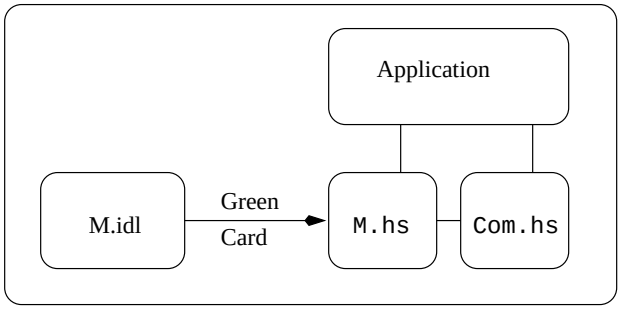

Figure 3: The big picture

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 2 gives the IDL description of the `IUnknown` interface, the interface of which every other is a superset. The 128 bit long constant is the GUID for the `IUnknown` interface.

4 Interfacing Haskell and COM

Our goal is to provide a convenient and type-secure interface between a Haskell program and the COM objects it manipulates. How could COM objects appear to the Haskell programmer?

Our approach, illustrated in Figure 3, is broadly conventional. We have built a tool, called Green Card, that takes an IDL module `M.idl`, and from it generates a Haskell module `M.hs`[1]. Object instances live in the C world (adding yet another level of indirection), and are accessed in the Haskell world using our previously developed foreign language interface to C[9]. Green Card automatically generates all required stub procedures and marshalling code to call C. The `M.hs` module, together with a library Haskell module `Com.hs`, is all that an application need import to access and manipulate all the COM objects described by `M`.

4.1 What Green Card generates

So what does the Haskell module `M` export?

- For each CLSID `Baz` in the IDL module, module `M` exports a value `baz` of type `ClassId`. This value represents the CLSID of class `Baz`. `ClassId` is an abstract type exported by `Com.hs`.
- For each IID `IFoo` in the IDL module, `M` exports:
  - A new, abstract, Haskell data type `IFoo`. Surprisingly, no operations are provided on values of type `IFoo`.
  - A value `iFoo` of type `Interface IFoo`. This value represents the IID for `IFoo`. `Interface` is an abstract type constructor exported by `Com.hs`.

  An interface pointer for an interface whose IID is `IFoo` is represented by a Haskell value of type `Com IFoo`. `Com` is an abstract type constructor exported by `Com.hs`.
- For each method `meth` in the interface `IFoo`, module `M` exports a Haskell function `meth` with the type:

  `meth :: `a_1` -> ... -> `a_n` -> Com IFoo -> IO `r

  Here, $a_1, \ldots, a_n$ are the argument types expected by `meth`, extracted from the method's IDL signature, and r is its result type. (If there are many results then `meth` would have a tuple result type `IO` $(r_1, \ldots, r_n)$.) The interface pointer is passed as the last argument for reasons we discuss later.

  Notice that `meth` cannot be invoked on any interface pointer whose type is other than `Com IFoo`, so the interface is type-secure.

  The result of `meth` has type "`IO` r" rather than simply "r" to signal that `meth` might perform some input/output. 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. All functions that can perform I/O have a result type of the form $IO\ \tau$. This so-called *monadic I/O* has become the standard way to do input/output in purely functional languages [8].

[1] In fact, rather than reading the IDL text directly, the tool interrogates the *type library* for `M`, a COM object generated by a Microsoft tool from the IDL. The Microsoft tool does all the parsing and type-checking of the IDL. The type-library object it produces is essentially a parse tree with methods that allow its clients to navigate the parse tree. The tool itself is written in Haskell and has been bootstrapped to generate the Haskell module to access type library components.

- The library module `Com.hs` provides a generic procedure `createInstance`:

```
createInstance :: ClassId
               -> Interface i
               -> IO (Com i)
```

  Like `CoCreateInstance`, it takes a CLSID and and IID, and returns an interface pointer. Unlike the C++ procedure `CoCreateInstance`, however, we use polymorphism to record the fact that the interface pointer returned "corresponds to" the IID passed as argument. This somewhat unusual use of polymorphism elegantly captures exactly what we want to say, and achieves type safety without having to resort to type casts as in C or Java.

  The IO type has an exception mechanism that is used to deal with the failure of `createInstance`.

- The library module `Com.hs` provides a generic procedure `queryInterface`:

```
queryInterface :: Interface j
               -> Com i
               -> IO (Com j)
```

  The first argument is the IID for the desired interface. The second is the interface on which we want to query for another interface. The result is an interface for the desired interface. Again, we use polymorphism to make sure that the interface that is returned by `queryInterface` (of type `Com j`) corresponds to the IID (of type `Interface j`) passed as the first argument.

- There are no programmer-visible procedures corresponding to `AddRef` and `Release`. Instead, when Haskell's garbage collector discovers that a value of type `Com i` is now inaccessible, it calls `Release` on the interface pointer it encapsulates. This is just a form of *finalization*, a well-known technique in which the garbage collector calls a user-defined procedure when it releases the store held by an object.

4.2 The Agent example

These points make more sense in the context of a particular example. Suppose we took the IDL description for Microsoft Agent. After being processed by Green Card, we would have a Haskell module `Agent.hs` that exports (among other things) the types, functions, and values given in Figure 4.

Microsoft Agent implements cartoon characters that pop up on the screen and talk to you. The animation is supported by an *agent server* whose CLSID is `agentServer`, and whose main interface is `IAgent`. Once we have created an agent server, we can load a character from a file, getting a `CharId` in reply. Now we can generate instances of that character using `getCharacter`, getting an interface pointer for the character in return[2]. Having got a character, we can make it talk a sentence by calling `speak`, or play a little animation by calling `play`.

Here is a complete example program:

```
module Main where
import Agent

main = comRun $
 do server <- createInstance
                 agentServer SERVER iAgent
    rob_id <- server # load "robby.acs"
    robby  <- server # getCharacter rob_id
    robby # moveTo centerScreen
    robby # show
    robby # speak "Hello world"
```

To make sense of this, we need to know the following Haskell lore:

- Left associative function application is written as juxtaposition. Thus `f a b` means "`f` applied to `a` and `b`". Right associative function application is written as `$`. Thus `f $ g a` means "`f` applied to `g a`".

- The function `#` is simply reverse function application.

```
(#) :: a -> (a->b) -> b
x # f = f x
```

  It is used here to allow us to write the interface pointer first in a method call, much as happens in an object oriented language. For example, `robby # speak "Hello"` means the same as `speak "Hello" robby`. It is for this reason that Green Card arranges that the interface pointer is the last parameter of each method call.

- The "`do`" notation is used to sequence a series of I/O-performing function calls. It is much more syntactically convenient than using the bind and unit functions of the monad, as the first papers about monadic I/O did [8, 10]. The statement `robby <- server # getCharacter rob_id` binds the result of performing the action `server # getCharacter rob_id` to the name `robby`.

Now we can read the example. The function `comRun` is exported by `Com.hs` and has type

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

It encapsulates a computation that accesses COM, preceding it with initialization and following it with finalization.

[2] It is quite common for COM calls to return interfaces. Here, `getCharacter` plays the role of `createInstance`, returning an interface to the new character. The interface may be have been created inside the agent server by a call to `CoCreateInstance` but that does not concern us.

```
module Agent where

      -- The Agent class
  agentServer :: ClassID

      -- The IAgent interface
  data IAgent = ...            -- Agent interface type
  iAgent :: Interface IAgent   --  ...and its IID

  type CharId = Int
  load         :: String -> Com IAgent -> IO CharId
  getCharacter :: CharId -> Com IAgent -> IO (Com IAgentCharacter)
  ...etc other methods of IAgent...

      -- The AgentCharacter interface
  data IAgentCharacter = ...  -- Ditto IAgentCharacter
  iAgentCharacter :: Interface IAgentCharacter

  type ReqId = Int
  play  :: String -> Com IAgentCharacter -> IO ReqId
  speak :: String -> Com IAgentCharacter -> IO ReqId
  wait  :: ReqId ->  Com IAgentCharacter -> IO ReqId
  ...etc other methods of IAgentCharacter...
```

Figure 4: Exports from module `Agent`

Next, the call to `createInstance` creates an instance of the agent server. The next two lines load the animation file "`robby.acs`" and create one instance of the character. The curious intermediate value, `rob_id`, is an artifact of the Agent server design, and not relevant here. In practice we would abstract from this design quirk and define a new function `createCharacter` as:

```
createCharacter :: String -> Com IAgent
                -> IO (Com IAgentCharacter)
createCharacter agent server =
 do a <- server # load agent
    server # getCharacter a
```

Finally, the character appears in the center of the screen and is asked to speak a phrase. All the `AddRef` and `Release` calls are handled implicitly.

5 Why use Haskell?

One can, of course, invoke COM objects from Visual Basic or C++. So is this paper of any interest to a VB or C++ programmer? We believe that it may be, as we argue in this section.

When we program our first example in C++ we see that we need to do a lot more bookkeeping:

```
void main ()
{
 IAgentServer* server    = NULL;
 IAgentCharacter* robby  = NULL;
 HRESULT hr; int reqid; int charid;

 hr = OleInitialize(NULL);
 if (checkHR(hr))
 {
   hr = CoCreateInstance( CLSID_AgentServer, NULL,
   CLSCTX_SERVER, IID_IAgentServer, &server );
   if (checkHR(hr))
   {
     hr = server->load( L"robby.acs", &charid );
     if (SUCCEEDED(hr))
     {
      server->getCharacter( charid, &robby );
     }
     if (checkHR(hr))
     {
       hr = robby->show( &reqid );
       hr = robby->speak( L"Hello world", &reqid );
       robby->Release();
     }
     server->Release();
   }
   OleUnitialize();
 }
}

int checkHR( HRESULT hr )
{
    if (FAILED(hr)) showError(hr);
    return (SUCCEEDED(hr));
}
```

The error checking clutters the code a lot and it is not at

all trivial to be sure to call `Release` or `OleUnitialize` when an error happens. Maybe that is the reason that most C++ programs just leave it out.

For simple scripts, there is hardly any difference between Haskell and say Visual Basic (or Java). Except for the declaration of the variable `Dim Robby` our Agent example looks similar. The COM initialization and finalization is done automatically as are the calls to `Release`.

```
Dim Robby
AgentControl.Connected = True
AgentControl.Characters.Load "Robby",
   ".....\robby.acs"
Set Robby = AgentControl.Characters("Robby")
...
Robby.Move (300,400)
Robby.Show
Robby.Speak "Hello, World!"
...
```

The real difference shows when we want to abstract from commonly occurring patterns in scripts.

5.1 Extending the characters' repertoire

The methods `play` and `speak` are rather limited. We would like to be able to define new, compound method, so that

```
robby # dancesAndSings
```

would make `robby` execute a sequence of `play` and `speak` actions. Here's how we can do that in Haskell:

```
type Action = Com IAgentCharacter -> IO ReqId

dancesAndSings :: Action
dancesAndSings agent =
 do agent # speak "La la la"
    agent # play "Dance"
```

Here we have defined the type `Action` as a shorthand to denote actions that can be performed by an agent (like `play "Dance"` or `dancesAndSings`).

In C++ or Java one could define `dancesAndSings` as the method of a class that inherits from `IAgentCharacter`, using implementation inheritance to arrange to call the character's own `play` or `speak` procedure. To us, it seems rather unnatural to introduce a type distinction between agents that can `dance` and `sing` and agents that can `danceAndSing`. Object oriented languages are good in expressing new objects as extensions of existing objects, functional languages are good in expressing new functions in terms of existing functions. In Visual Basic we could certainly define a procedure like `dancesAndSings`, but than we could only call it using a different syntax than native class methods.

```
Sub DancesAndSings (Byref Agent)
  Agent.Speak ("La la la")
  Agent.Play  ("Dance")
End Sub
...
Robby.Speak ("Hello")
DancesAndSings (Robby)
...
```

If the sequence of actions a particular agent has to perform gets long, it becomes a bit tiresome writing all the "`agent #`" parts, so we can rewrite the definition as a little script, like this:

```
dancesAndSings :: Action
dancesAndSings agent =
   agent # sequence [speak "La la la", play "Dance"]
```

where `sequence` is a re-usable function that executes a list of actions from left to right:

```
sequence :: [Action] -> Action
sequence [a]    agent = agent # a
sequence (a:as) agent =
   do agent # a; sequence as agent
```

Notice that the type of the first argument of `sequence` is a *list* of *functions* that return *I/O performing computations*. The ability to treat functions and computations as first-class values, and to be able to build and decompose lists easily, has a real payoff. In Java, C++, or VB it is much harder to define custom control structures such as `sequence`. For example in Java 1.1 one would use the package `java.lang.reflect` to reify classes and methods into first class values, or use the Command pattern [5] to implement a command interpreter on top of the underlying language. Note that in our case `sequence [...]` is another composite method on agents, just as `dancesAndSings`, and is called in exactly the same way as a native method.

The low cost of abstraction in Haskell is even more convincing when we define a family of higher-order functions to ease moving agents around the screen. First we define a function `movePath` as:

```
type Pos  = (Int,Int)

movePath :: [Pos] -> Action
movePath path agent =
 agent # sequence [ moveTo pos | pos <- path ]
```

Function `movePath path robby` moves agent `robby` along all the points in the list `path`. In Visual Basic (or Java) we can define a similar function quite easily as well by using the built-in `For ... Each ...Next` control structure:

```
Sub MovePath (Byref Agent, Byref Path)
  For Each Point In Path
     Agent.MoveTo (Point)
  Next point
End Sub
```

However, in Haskell we don't have to rely on foresight of the language designers to built in every control structure we might ever need in advance, since we can define our own

custom control structures on demand. Lazy evaluation and higher order functions are essential for this kind of extensibility [7].

We can use function `movePath` to construct functions that move an agent along more specific figures, such as squares and circles:

```
moveSquare :: Pos -> Int -> Action
moveSquare (x,y) width agent =
 agent # movePath square
  where
    w = width `div` 2
    square = [ (x-w,y-w), (x+w,y-w)
             , (x+w,y+w), (x-w,y+w)
             , (x-w,y-w)
             ]

moveCircle :: Pos -> Int -> Action
moveCircle (x,y) radius agent =
 agent # movePath circle
  where
    circle = [ ( x + (radius*cos t)
               , y + (radius*sin t)
               )
             | t <- [0,pi/100..pi]
             ]
```

By re-using `sequence` and `movePath` we were able to define `moveSquare` and `moveCircle` very easily. Because Haskell uses lazy evaluation, the lists of points are generated on demand and therefore never completely in memory.

5.2 Synchronization

The Agent server manages each character as a separate, sequential process, running concurrently with the other characters. Suppose we want one character to sing while the other dances, we just write:

```
do erik  # sings
   simon # dances
```

It looks as if these take place sequentially, but actually they are done in parallel. Each character maintains a queue of requests it has got from the server and performs these in sequence. Hence a call such as `erik # sings` returns immediately, while `erik` is still singing and then makes `simon` dance in parallel.

Now suppose we want `daan` to do something else only when both `erik` and `simon` have terminated; how can we ask the Agent server to do that? The answer is that every `Action` returns a *request ID*, of type `ReqId`, on which any character can `wait`, to synchronize on the completion of that request. Thus:

```
do erikDone  <- erik # sings
   simonDone <- simon # dances
   daan # wait erikDone; daan # wait simonDone
   daan # speak "They're both done"
```

You may imagine that in a complex animation it can be complicated to get all these synchronizations correct. We might easily wait for the wrong request ID, or get deadlocked, or whatever. What we would like to be able to do instead is to say something like:

```
(erik # sings) <|> (simon # dances)
<*>
(daan # speak "They're both done")
```

Here `<*>` is an infix operator used to compose two animations in sequence, and `<|>` composes two animations in parallel. Since all the synchronization is now implicit, it is much harder to get things wrong. We can now say what we want, since we have abstracted away from the details how we have to encode all the low-level synchronization between agents.

How can we program these "animation abstractions" in Haskell?

To perform two animations in sequence, we need to wait until all actions in the first animation are performed before we can start the second. If we assume that an animation returns the request-id of the very last action it performs, we can wait for that one and be sure that all other actions in that animation are also completed. In order to be able to make an animation wait for a request-id, we need to know all characters that will perform in that animation — its *"cast"*. Hence, we represent animations by a pair of an action that returns a request-id, and the cast for that action:

```
type Anim = (IO ReqId, [Com IAgentCharacter])
```

Using type `Anim`, we could (erroneously) try to define sequential composition of two animations as follows:

```
(action1, cast1) <*> (action2, cast2) =
 (action, cast1 `union` cast2)
 where
   action =
    do r1 <- action1
       cast2 `waitFor` r1
       action2
```

Unfortunately, this solution does not work because we can get a deadlock when an agent is part of both animations, in which case it could be waiting for itself to terminate. We therefore take the difference (`\\`) between the casts involved in the two animations.

A more subtle problem occurs when more than two animations are composed in sequence. Suppose we compose three animations thus, `(s1 <*> s2) <*> s3`, and suppose that agent `daan` plays a role in `s1` and `s3` but not `s2`. The deadlock-avoidance device means that `daan` will not wait for `s2` to conclude before starting whatever actions are scripted for him in `s3`. The solution is a little counter-intuitive: in the composition `s1 <*> s2`, make the cast of `s1` who are not involved in `s2` wait for the the cast of `s2` to finish.

Our final (and correct) version of `<*>` will therefore be:

```
(<*>) :: Anim -> Anim -> Anim
(action1, cast1) <*> (action2, cast2) =
 (action, cast1 `union` cast2)
 where
   action =
    do reqid1 <- action1
       (cast2 \\ cast1) `waitFor` reqid1
       reqid2 <- action2
       (cast1 \\ cast2) `waitFor` reqid2
```

The operation `waitFor cast reqid` makes every agent `a` in its input list `cast` wait on the given request-id `reqid`. Function `as `waitFor` reqid` always returns `reqid`.

```
waitFor :: [Com IAgentCharacter] -> ReqId
        -> IO ReqId
[]     `waitFor` reqid = return reqid
(a:as) `waitFor` reqid =
 do a # wait reqid
    as `waitFor` reqid
```

The definition of parallel composition is now easy. We let all the agents of the second animation wait for the first animation to complete and the other way around. Note the nice duality in the implementation of the sequential and parallel combinator: we just swap the middle two statements.

```
(<|>) :: Anim -> Anim -> Anim
(action1, cast1) <|> (action2, cast2)  =
 (action, cast1 `union` cast2)
 where
   action =
    do reqid1 <- action1
       reqid2 <- action2
       (cast2 // cast1) `waitFor` reqid1
       (cast1 // cast2) `waitFor` reqid2
```

In about 20 lines of code we have a very clear definition and implementation of two non-trivial combinators. Using the properties of a pure lazy language we can use equational reasoning to prove various of laws that we expect to hold for the combinators:

```
x <*> (y <*> z)  =  (x <*> y) <*> z
x <|> (y <|> z)  =  (x <|> y) <|> z
        x <|> y  =  y <|> x
```

Proving properties like these is not just a technical nicety! As we have already seen, obtaining correct synchronization among the characters is somewhat subtle, and conducting proofs of properties like these can reveal subtle bugs. This happened to us in practice: when proving the associative law for `<*>`, we discovered that our previous implementation was incorrect in a subtle way.

6 What next?

So far we have described how we may access COM objects from a Haskell program. The obvious dual is to encapsulate a Haskell program as a COM object. We plan to do this next, but there are some interesting new challenges. Chief among these is that a COM object implemented in Haskell must be supported by a Haskell run-time system and garbage-collected heap. While the code might be shared, we would prefer not to create a separate heap for each object; remember a COM object might represent a rather lightweight thing like a button or a scroll-bar. Instead, we would like all the Haskell objects in a process to share the same RTS and heap.

Besides encapsulating a Haskell *program* as a COM object, we also plan to encapsulate a Haskell *interpreter* as a COM object, which implements the `IScriptEngine` interface. This allows us to use Haskell programs to script interactive Web pages

```
<SCRIPT LANGUAGE="HaskellScript">
   do yes <- confirm ("Do you like Haskell?")
      document#write ( if yes then "I knew it!"
                       else "Are you sure?"
                     )
</SCRIPT>
```

or as embedded macro language for MS Office applications such as Word and Excell. Similar implementations already exist for Visual Basic, Java Script, Perl and Phyton.

7 Summary

The theme of this paper is that it is not only *possible* to script COM components in Haskell, but also *desirable* to do so.

We have described a simple way to incorporate COM objects into Haskell's type system, making use of polymorphism to enforce the connection between an IID and the interface pointer returned by `queryInterface`.

We have also shown how one can use higher-order functions, and first-class computations (that is, values of type `IO` τ), to define powerful new abstractions. In the Agent example, we built a little custom-designed sub-language, or combinator library, for expressing parallel behavior. The implementation of the combinators is terse enough that we were able to perform simple algebraic proofs of their properties.

All of this can doubtless be done in any programming language. Our only claim here is that higher-order, typed, functional languages make the job considerably easier.

Acknowledgments

We acknowledge gratefully the support of the Oregon Graduate Institute during our sabbaticals, funded by a contract with US Air Force Material Command (F19628-93-C-0069). Machines and software were supported in part by gifts from Microsoft Research.

References

[1] Kraig Brockschmidt. *Inside OLE (second edition).* Microsoft Press, 1995.

[2] David Chappel. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[3] Adam Denning. *ActiveX Controls Inside Out (second edition).* Microsoft Press, 1997.

[4] J. Peterson (editor). Report on the programming language HASKELL version 1.4. Technical report, http://www.haskell.org/, April 6 1997.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[6] Object Management Group. *The Common Object Request Broker: Architecture and Specification (revision 1.2).* Object Management Group, 1993. OMG Document Number 93.12.43.

[7] John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[8] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL 20*, pages 71–84, 1993.

[9] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green card: a foreign-language interface for haskell. In *Proc. Haskell Workshop*, 1997.

[10] SL Peyton Jones and J Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[11] Microsoft Press. *Automation Programmers Reference*, 1997.

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

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

A Outline of proof of associativity of <*>

In order to prove that <*> is associative, we make some assumptions on the agent implementation.

The first assumption is that the call `as 'waitFor' r` behaves like the identity function with a side effect of letting all agents in `as` wait for request id `r`. We assume that waitFor has no other visible side effect. It then follows that waitFor distributes over set union:

```
(as `union` bs) `waitFor` r =
    do as `waitFor` r; bs `waitFor` r
```

or equivalently that waiting is commutative and idempotent:

```
do as `waitFor` r; as `waitFor` r =
       as `waitFor` r
do as `waitFor` r; bs `waitFor` r =
    bs `waitFor` r; as `waitFor` r
```

The next law states that agents don't have to wait twice in a row:

```
 as `waitFor` r1;
(as `union` bs) `waitFor` r2 =
    (as `union` bs) `waitFor` r2
```

When there is no interaction between the set of agents that are waiting and the cast of a subsequent action then waiting can be delayed.

```
as `waitFor` r1; r2 <- action =
    r1 <- action; as `waitFor` r2
```

Using the above laws plus standard set theory, it follows that <*> is associative.

```
(action1,c1) <*> ((action2,c2) <*> (action3,c3))
```

First, we unfold the definition of <*>

```
do r1  <- action1
   (c2 `union` c3)\\c1  `waitFor` r1
   r23 <- do r2 <- action2
             c3\\c2 `waitFor` r2
             r3 <- action3
             c2\\c3  `waitFor` r3
   c1\\(c2 `union` c3) `waitFor` r23
```

Next we flatten the sequence of actions

```
do r1 <- action1
   c2\\c1  `waitFor` r1
   c3\\(c1 `union` c2)  `waitFor` r1
   r2 <- action2
   c3\\c2  `waitFor` r2
   r3 <- action3
   r23 <- (c2\\c3) `waitFor` r3
   c1\\(c2 `union` c3) `waitFor` r23
```

We rearrange the statements by applying the various swap laws

```
do r1 <- action1
   c2\\c1  `waitFor` r1
   r2 <- action2
   c1\\c2  `waitFor` r2
   c3\\(c1 `union` c2)  `waitFor` r2
   r3 <- action3
   c2\\c3  `waitFor` r3
   c1\\(c2 `union` c3)  `waitFor` r3
```

and introduce nesting again

```
do r12 <- do r1 <- action1
```

```
            c2\\c1 `waitFor` r1
            r2 <- action2
            c1\\c2 `waitFor` r2
    c3\\(c1 `union` c2) `waitFor` r12
    r3 <- action3
    (c1 `union` c2)\\c3 `waitFor` r3
```

so that finally, we can fold the definition of `<*>`

```
((action1,c1) <*> (action2,c2)) <*> (action3,c3)
```